Rumoca: Towards a Translator from Modelica to Algebraic Modeling Languages

Micah K. Condie[1] Abigaile Woodbury[1] James M. Goppert[1] Joel A.E. Andersson[2]

[1]School of Aeronautics and Astronautics, Purdue University, United States,
{condiem,awoodbu,jgoppert}@purdue.edu
[2]Freelance software developer and consultant, United States & Finland, joel@jaeandersson.com

Abstract

We present Rumoca, a translator written in Rust that forms the basis of a symbolic toolchain, automatically converting Modelica models into a variety of target algebraic modeling languages. Rumoca is demonstrated on three models and translated into two different algebraic representations: CasADi and SymPy. Designed for generalizability, Rumoca has the potential to accommodate increasingly complex Modelica models and additional target languages.

Keywords: Modelica, symbolic computation, algebraic modeling, cyber-physical systems, Model Translation, CasADi, SymPy

1 Introduction

Cyber-physical systems (CPSs) are designed using a variety of modeling languages and tools, each with its own advantages (Amrani et al. 2021). However, interoperability between these tools remains limited. Porting models across platforms often requires manual conversion or unverifiable code generation techniques, such as those based on large language models (LLMs), which is both time-consuming and error-prone (Liu et al. 2020). To address this, we present Rumoca, a translator written in Rust that converts Modelica models into various algebraic modeling backends, enabling symbolic manipulation, automatic differentiation, and optimization (Jakob Andersson et al. 2019).

The long-term vision for Rumoca is to provide a flexible, extensible compiler infrastructure for hybrid systems modeling and analysis. It targets full coverage of a flattened Modelica representation—stripped of class hierarchies as defined later in this paper—and currently supports SymPy and CasADi backends. By exposing a lightweight abstract syntax tree (AST) and a modular templating engine, Rumoca makes it easy to develop additional code generation backends.

1.1 Modeling and Compiler Languages

Modelica was selected as the source language due to its concise semantics for modeling CPSs and its precise mapping to differential-algebraic equations (DAE) as defined in the language standard (Henriksson et al. 2011). These characteristics make it more verifiable than general-purpose languages like Python and C++, while availability of graphical editors and a large user community further support its adoption.

Rumoca is implemented in Rust, which offers significant benefits for compiler construction. The language provides strong static typing and memory safety without garbage collection, helping to eliminate entire classes of bugs while maintaining runtime performance. Modern features like pattern matching, algebraic data types, and expressive traits support the creation of robust and maintainable translation infrastructure (Klabnik and Nichols 2019; Matsakis and Klock 2014; Jung et al. 2018).

1.2 Target Languages

In designing Rumoca, CasADi and SymPy were selected as the primary languages for translated Modelica models. These languages were chosen because of their strengths in symbolic and numerical computation and optimization.

CasADi is a symbolic framework for algorithmic differentiation and numeric optimization, particularly well-suited for control and estimation in dynamic systems (Jakob Andersson et al. 2019). It represents expressions as directed acyclic graphs (DAGs), which allows for efficient symbolic manipulation and code generation. This structured representation offers significant advantages for optimization tasks compared to black-box models, such as those commonly encapsulated in Functional Mock-up Units (FMUs). CasADi is widely used in optimal control, parameter estimation, and nonlinear programming, especially in embedded applications where runtime performance and gradient computation are critical. By translating Modelica models into CasADi expressions, Rumoca enables users to perform gradient-based optimization, sensitivity analysis, and simulation leveraging high-performance solvers.

A key advantage of using CasADi is its close integration with numerical solvers like IPOPT (Biegler et al. 2009) and C code generation. This makes it highly suitable for deploying optimized controllers or estimators on embedded platforms. A disadvantage of CasADi is its lack of object-oriented features, which makes it difficult to scale to larger, more sophisticated models. However, since object-oriented design is a strength of Modelica, CasADi

is a strong candidate for serving as the backend of the symbolic toolchain enabled by Rumoca.

SymPy, in contrast, is a general-purpose computer algebra system written entirely in Python (Meurer et al. 2017). It supports a broad range of symbolic operations including simplification, equation solving, symbolic differentiation and integration, and matrix algebra. SymPy is especially useful for verifying model structure, performing symbolic manipulations, and producing readable mathematical expressions. These capabilities make SymPy an ideal target when the goal is to inspect or manipulate equations algebraically.

Although SymPy lacks the high-performance optimization and solver integration features of CasADi, it excels in educational, research, and verification settings. SymPy's strength lies in its ability to express mathematical models symbolically and perform analytical transformations, including the computation of Jacobians, Laplace transforms, or simplification of symbolic DAE systems. Its integration with Jupyter and Python's scientific stack makes it highly accessible and extendable. As an immediate application of the SymPy backend, we intend to leverage these computational features to construct reachability proofs.

1.3 Related Work

Numerous tools exist for compiling or translating Modelica models, each with distinct design goals. Rumoca distinguishes itself by acting as a *translator*–focused on high-level symbolic representation and model interoperability–rather than as a compiler generating low-level machine code.

The most mature open-source option is OpenModelica, which supports the Modelica Standard Library and offers a graphical user interface via OMEdit ((OSMC) 2025). While well-suited for simulation and code generation, OpenModelica does not currently support symbolic backends such as CasADi or SymPy. Since the OpenModelica compiler does support multi-code generation backends, it is possible it could be extended to these languages. Rumoca, however, offers unique advantages by being implemented in Rust, having algebraic modeling languages in mind.

Marco is a high-performance Modelica compiler written in C++ and based on LLVM (Agosta et al. 2023). It targets efficient simulation of large-scale models, prioritizing speed and robustness. However, Marco is focused on generating C code and does not offer translation into symbolic frameworks, limiting its utility for applications involving algebraic transformation or formal analysis.

In constrast to traditional Modelica compilers that focus on simulation and code generation, Rumoca is designed as a translator–targeting symbolic and algebraic frameworks. Its purpose is not to compete on execution speed or complete language coverage, but to facilitate workflows involving optimization, model verification, and hybrid system analysis. We envision a workflow that utilizes OpenModelica or Marco, by using these compilers to convert Modelica models to an intermediate representation that could be handled by Rumoca. The clear candidate would be the ongoing effort to define a BaseModelica language specification, which is a simplified subset of Modelica.

While Rumoca may not be able to handle a complex model directly, the model could first pass through a compiler such as Marco or OpenModelica, removing its object-oriented structure, and produce the model in BaseModelica. The BaseModelica version could then be processed by Rumoca. This proposed workflow is shown in Figure 1.

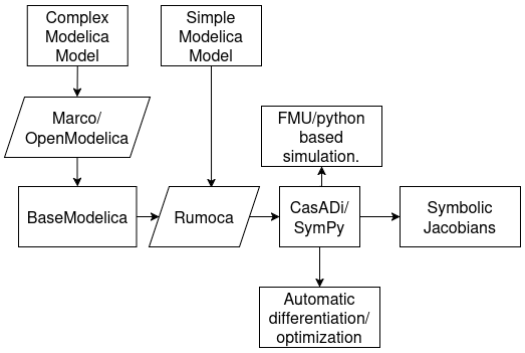

Figure 1. Proposed workflow utilizing compilers for generation of BaseModelica. Rumoca is a translator that takes simple Modelica models to target algebraic modeling languages. Complex models could first be passed through a more extensive compiler, removing the object oriented structure, before being processed by Rumoca.

Another notable effort is Pymoca, an open-source translator written in Python and based on ANTLR. It shares similar goals with Rumoca and provides a Pythonic interface for model transformation (Contributors 2025). Pymoca benefits from accessibility and ease of use, especially for Python developers. However, it suffers from significant performance limitations due to Python's speed and the use of un-typed ASTs. The lack of strong typing and the slow parsing pipeline–where the source is first converted into a parse tree before AST generation–makes Pymoca less suitable for handling large-scale models efficiently.

1.4 Overview

The remainder of the paper is structured as follows. Section 2 explains the details of Rumoca, including modifications to the Parol parser, structure of the AST, and use of Jinja2 templates for the generator code. Section 3 describes some changes that were made to the CasADi library to more easily accommodate the Modelica structure. Section 4 demonstrates Rumoca on three example systems, converting them from Modelica to Sympy and CasADi. Section 5 provides a summary and direction for future work on the Rumoca translator.

DOI 10.3384/ecp2181009

2 Rumoca Details

The compiler pipeline can be understood in terms of four components. First, the automatically generated parse tree is overwritten to a customized IR format. Second, this AST is flattened, removing class hierarchies. Third, the flattened IR is lowered into a DAE representation, consistent with the semantics defined by the Modelica specification. Finally, the DAE system is rendered into the target language using a template engine.

2.1 Parser

For parsing, Rumoca is using the Parol parser (Singer 2025). Parol is a Rust based parser generator, similar to ANTLR. Given an EBNF grammar file, Parol can automatically generate a parser. While we evaluated other parser generators such as ANTLR and LALRPOP, we found that Parol was best suited for the project. ANTLR handles EBNF grammar well, but does not have a native Rust implementation, and lacks the capability to significantly modify the automatically generated parser code. ANTLR does however support (LR*) grammar with arbitrary look-ahead which simplifies grammar handling.

The LALRPOP parser generator, which handles LR(1) grammar, we found to be limiting, and the compilation times for the compiler slowed our development cycle. LALRPOP did provide an easy mechanism for modifying the generated parser tree at runtime so that a second pass over the parser tree was not generated to obtain an abstract syntax tree.

Finally, Parol was selected as it had the advantage of supporting EBNF grammar and allowing for customization of the parse tree at parse time. When customization of the parse tree is necessary to obtain a desirable abstract syntax tree to be used as an internal representation, it is useful to selectively re-implement the parse tree generation for specific language elements. The automatically generated parsing functions may then be leveraged when customization is not necessary. Parol has a fast compile time, similar to ANTLR, and is a LL(k) parser. LL(k) means that the EBNF grammar is analyze for the largest look-ahead necessary. We were able to implement the Modelica language specification with a look-ahead of 3 tokens (k = 3). In order to simplify the implementation of the Modelica language parser, we requested several features to be added to Parol, and they were quickly implemented.

2.2 Language Coverage

While the Parol parser automatically generates a parse tree for the entire Modelica language grammar, the steps converting it to the intermediate representations require significant customization. For this reason, Rumoca only supports a portion of the Modelica language. Independently, there is an ongoing effort to define a BaseModelica language specification, which is a simplified subset of Modelica. It is our goal to have Rumoca work for the entirety of the BaseModelica features. Table 1 shows a preliminary draft of the features that may be included in BaseModelica, and which are currently supported by Rumoca. The features listed were taken from the BaseModelica meeting notes (Modelica Association 2023). We envision that compilers such as OMC or Marco will compile to BaseModelica, so that by chaining the tools any Modelica model could be handled by Rumoca.

Table 1. Rumoca support for selected Modelica features

Modelica Feature	*Supported*
Scalar types (e.g., `Real`, `Integer`)	Yes
Record and enum types	No
Arrays (decl. & indexing)	Partial
Component decl.	Yes
Equations and algorithms	Yes
Source location metadata	No
Documentation strings	Yes
Vendor-specific annotations	No
All variabilities (no param eval)	Yes
Params treated as const	Yes
Preserve values of inlined constants	Yes
Relaxed record field access	No
Relaxed array subscripting	Partial
Naming of intermediate vars	No
Alias expressions (ref & sign)	No
Used function declarations	Yes

In addition to the features listed, we have added support for simple class structures. This is because we intend to use Rumoca for hierarchical reachability proofs which require nested components.

2.3 Generator Code

Jinja2, a widely used text templating engine for Python, is employed to construct the generator code. With millions of downloads per month and integration into major frameworks such as Flask and Ansible, Jinja2 benefits from a large and active user base (Index 2024; Projects 2024; Red Hat 2024). In Rumoca, each Jinja template defines how the symbolic expressions described in the AST are rendered into source code. Due to the common DAE structure of the AST, templates can be created and modified for specific target languages with relative ease. This gives Rumoca the advantage of being extendable to a variety of backends with minimal effort and expertise. This allows for end-users to create custom backends. Examples and documentation for writing custom templates are available online.[1]

An example of how a template defines the handling of model parameters is shown below. Listing 1 presents a portion of the template for rendering the CasADi repre-

[1] `https://github.com/CogniPilot/rumoca/blob/main/tests/templates`

sentation, while Listing 2 displays the corresponding rendered code.

Listing 1. Jinja2 template for parameter handling

```
{%- elif var == 'p' %}
    # Declare {{ var }}
    {% for c in dae[var] -%}
    {{ c.name }} = dae.add('{{ c.name }}',
        'parameter', 'tunable', dict(start
        = {{ render_expression(c.start) }})
        )
    {% endfor -%}
```

Listing 2. Rendered code for parameter declarations

```
# Declare p
e = dae.add('e', 'parameter', 'tunable',
    dict(start = 0.8))
h0 = dae.add('h0', 'parameter', 'tunable',
    dict(start = 1.0))
```

3 CasADi Additions

In this work we leverage and extend CasADi's DaeBuilder class (Joel Andersson 2024), which can be used for both import of Modelica in both symbolic form and in the form of exported FMUs adhering to FMI 2.0 and 3.0. In the recent CasADi 3.7 release, the DaeBuilder class was refactored to use a representation of model variables that closely resembles the representation of model variables in the FMI standard. The same release also saw a new refactored high-level interface for symbolically defining model equations, now including model equations with event dynamics, compatible with the recently added support for analytic derivative calculations for dynamic systems with events (Joel Andersson and Goppert 2025). Another important update is an extension of CasADi's support for FMI export, which can now be used for systems with events. We refer to the updated CasADi user guide for details on this support.

4 Examples

This section demonstrates the use of Rumoca on three representative examples: a simple bouncing ball, a mass-spring-damper system, and a more complex quadrotor model. For each case, Rumoca takes the corresponding Modelica model and translates it into either a SymPy or CasADi representation, depending on the selected backend template. These examples illustrate the class of systems that Rumoca currently supports, ranging from basic hybrid dynamics to nonlinear multibody systems.

In addition to translation, further analyses are performed to highlight some of the benefits of the target symbolic languages. Specifically, an optimal disturbance attack is computed for the mass-spring-damper model, leveraging CasADi's directed acyclic graph (DAG) structure and automatic differentiation. For the quadrotor, the Jacobian matrix is extracted using SymPy's symbolic differentiation capabilities, which enable exact analytical expressions for sensitivity analysis and control design.

Rumoca is accessed through a command-line interface. Listing 3 shows the help menu output. As shown, Rumoca requires two inputs: a Modelica model and a code generation template specifying the target language. Each example uses a distinct Modelica file and a corresponding template—either for SymPy or CasADi—to produce the desired output.

Listing 3. Rumoca Help Menu

```
Rumoca Modelica Translator

Usage: rumoca [OPTIONS] --template-file <
    TEMPLATE_FILE> --model-file <MODEL_FILE
    >

Options:
  -t, --template-file <TEMPLATE_FILE>  The
      template
  -m, --model-file <MODEL_FILE>        The
      model file to compile
  -v, --verbose
      Verbose output
  -h, --help
      Print help
  -V, --version
      Print version
```

4.1 Bouncing Ball

As a first example, consider the classic bouncing ball model. Listing 4 shows this in Modelica.

Listing 4. Bouncing Ball model in Modelica

```
model BouncingBall "The 'classic' bouncing
    ball model"
  parameter Real e=0.8 "Coefficient of
      restitution";
  parameter Real h0=1.0 "Initial height";
  Real h = 5.0 "Height";
  Real v "Velocity";
  Real z;
equation
  z = 2*h + v;
  der(h) = v;
  der(v) = -9.81;
  when h<0 then
    reinit(v, -e*pre(v));
  end when;
end BouncingBall;
```

The bouncing ball model above is processed by Rumoca using the CasADi template, which generates a corresponding Python script. Listing 4.1 shows the command used to run this transformation via the command-line interface.

```
rumoca BouncingBall.mo -t casadi.jinja >
    bouncing_ball_ca.py
```

This generated script, here called "bouncing_ball_ca.py", contains a class with several methods. A portion of this code is shown in Listing 5.

Listing 5. A portion of the casadi code generated by rumoca (bouncing_ball_ca.py)

DOI 10.3384/ecp2181009

```
def c0():
    return (h < 0.0)

# ============================================
# Define reset functions: fr
def c0_fr():
    return dae.reinit("v", -((e * pre_v)))
dae.when(c0(), [c0_fr()])
# ============================================
# Declare x_dot
der_h = dae.der(h)
der_v = dae.der(v)
# ============================================
```

The example in Listing 6 demonstrates how to simulate the bouncing ball using the generated CasADi class, the result of which, is Figure 2 .

Listing 6. Simulation of the bouncing ball model using CasADi backend

```
import bouncing_ball_ca
import matplotlib.pyplot as plt

model = bouncing_ball_ca.Model("
    bouncing_ball")
tgrid, res = model.simulate(t0=0, tf=8, dt
    =0.01)

plt.plot(tgrid, res['xf'].T, label=model.
    dae.x())
plt.grid()
plt.legend()
plt.show()
```

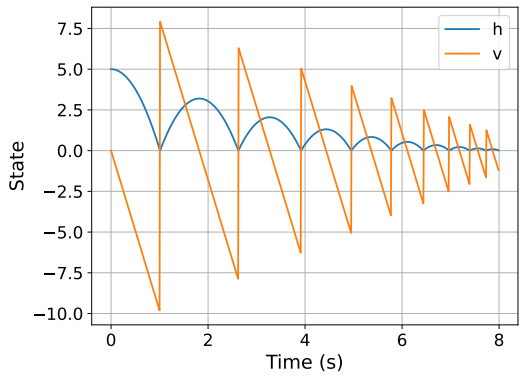

Figure 2. Height (m) and velocity (m/s) of a bouncing ball as a function of time. The simulation was done using the automatically generated CasADi script.

Consider now the example of using Rumoca on the bouncing ball model with the SymPy template. Here, the generated code is a script "bouncing_ball_sympy.py." As with the CasADi generated script, this code contains a class with several methods. A portion of this code is shown in Listing 7.

Listing 7. A portion of the SymPy code generated by rumoca ("bouncing_ball_sympy.py")

```
# Define Reset Functions: fr
def __fr_c0(x):
    pre_h, pre_v = self.x
    h, v = self.x
    v = -((e * pre_v))
    return [
        h,
        v]
self.fr_c0 = sympy.lambdify([self.x, self.p
    ], __fr_c0(self.x))

#
    ============================================

# Define Condition Update Function: fc
self.fc = sympy.Tuple(*[
    (h < 0.0)])
self.f_c = sympy.lambdify(
    args=[self.time, self.x],
    expr=self.fc,
    modules=['numpy'])
```

Listing 8 demonstrates running the bouncing ball simulation with the SymPy script, adjusting the initial height and velocity as well as the coefficient of restitution parameter. The corresponding plot can be seen in Figure 3.

Listing 8. Simulation of the bouncing ball model using SymPy backend

```
import bouncing_ball_sympy
import matplotlib.pyplot as plt

model = bouncing_ball_sympy.Model()
model.solve()
model.x0['h'] = 10
model.x0['v'] = 5
model.p0['e'] = 0.7
res = model.simulate(t0=0, tf=8, dt=0.01)

plt.plot(res['t'], res['x'].T, label=model.
    x0.keys())
plt.legend()
plt.grid()
```

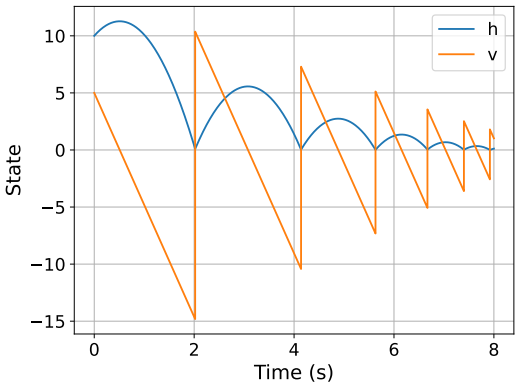

Figure 3. Height (m) and velocity (m/s) of a bouncing ball as a function of time. The simulation was done using the automatically generated SymPy script.

4.2 Mass-Spring-Damper

As an example of leveraging CasADi optimization, consider the simple mass spring damper system in Modelica

with a disturbance force shown in Listing 9.

Listing 9. Mass-spring-damper system

```
model msd "Mass spring damper"
  parameter Real k=1.0 "Coefficient of
      spring";
  parameter Real c=1.0 "Coefficient of
      damper";
  parameter Real m=1.0 "mass";
  Real x = 0.0 "Position";
  Real v "Velocity";
  input Real u "Disturbance";
equation
  der(x) = v;
  der(v) = -(k/m)*x - (c/m)*v - (1/m)*u;
end msd;
```

Once the model is processed by Rumoca using the CasADi template, the system can be optimized using familiar CasADi features. Consider the problem of optimizing a bounded disturbance for the MSD system such that it causes the mass to deviate as far as possible from equilibrium. This problem is solved using CasADi optimization tools, and the result is shown in Figure 4. This was done with the system only ever written in Modelica.

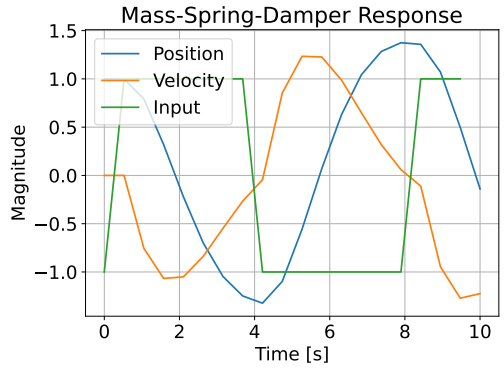

Figure 4. The optimized bounded control input to maximize disturbance from equilibrium. This was found using CasADi tools, having only written the system in Modelica.

4.3 Quadrotor

Rumoca is also capable of handling more complex models, including some with a hierarchical structure. The quadrotor model shown in Listing 10, extends a rigid body model and makes use of a separate motor model. Automatic generation of the corresponding CasADi and SymPy code are shown and demonstrate handling control inputs and using Sympy to linearize the model.

Listing 10. Quadrotor model definition in Modelica

```
model Quadrotor
    extends RigidBody6DOF;
    parameter Real l = 1.0;
    parameter Real mix_a = 1;
    parameter Real mix_e = 1;
    parameter Real mix_r = 10;
    parameter Real mix_t = 32.0;
    Motor m_1, m_2, m_3, m_4;
    input Real a "aileron";
    input Real e "elevator";
    input Real r "rudder";
    input Real t "throttle";
equation
    // body forces
    F_x = -(m*g)*sin(theta);
    F_y = (m*g)*sin(phi)*cos(theta);
    F_z = (m*g)*cos(phi)*cos(theta) -
          (m_1.thrust + m_2.thrust + m_3.
              thrust + m_4.thrust);

    // body moments
    M_x = l*(-m_1.thrust + m_2.thrust - m_3
        .thrust + m_4.thrust);
    M_y = l*(-m_1.thrust + m_2.thrust + m_3
        .thrust - m_4.thrust);
    M_z = m_1.moment + m_2.moment - m_3.
        moment - m_4.moment;

    // motor equations
    m_1.omega_ref = t*mix_t - a*mix_a + e*
        mix_e + r*mix_r;
    m_2.omega_ref = t*mix_t + a*mix_a - e*
        mix_e + r*mix_r;
    m_3.omega_ref = t*mix_t - a*mix_a - e*
        mix_e - r*mix_r;
    m_4.omega_ref = t*mix_t + a*mix_a + e*
        mix_e - r*mix_r;
end Quadrotor;
```

4.3.1 CasADi Quadrotor Simulation

Upon running Rumoca for the quadrotor model using the CasADi template, the "quadrotor_casadi.py" python script is generated. Listing 11 demonstrates how the control input can be adjusted and the quadrotor simulated. In this case, the throttle is set to a constant value. The resulting position of the quadrotor upon running the CasADi simulation is shown in Figure 5.

Listing 11. CasADi simulation setup for the quadrotor model

```
model = quadrotor_casadi.Model("quad")
states = model.dae.x()
params = model.dae.p()

x0 = model.dae.start(states)
param_vals = model.dae.start(params)

# Set control input: aileron, elevator,
    rudder, throttle
inputs = np.array([0, 0, 0, 0.5])

# Set initial height of quadrotor to 10 m.
x0[states.index('h')] = 10
tgrid, res = model.simulate(t0=0, tf=10, dt
    =0.01, f_u=inputs, x0=x0)
```

4.3.2 SymPy Quadrotor Linearization

Rumoca's SymPy output enables the generation of symbolic expressions for the Modelica model. Eq, displays a portion of the 16×16 linearized state matrix for the quadrotor automatically generated by Sympy from the

DOI
10.3384/ecp2181009

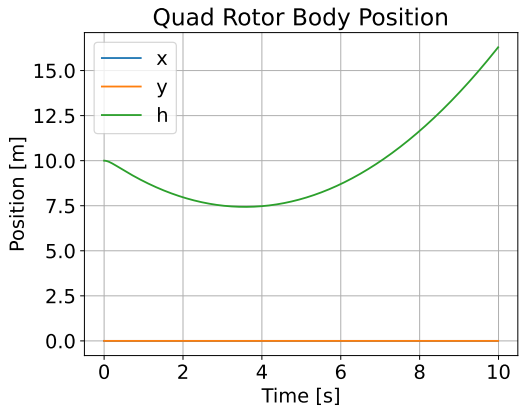

Figure 5. Quadrotor position from CasADi simulation.

Modelica model, illustrating the power of using SymPy as a back-end.

$$\begin{array}{ccc} \frac{J_x J_{xz} Q - J_{xz} J_y Q + J_{xz} J_z Q}{\Lambda} & \frac{J_x J_{xz} P - J_{xz}^2 R - J_{xz} J_y P + J_{xz} J_z P + J_y J_z R - J_z^2 R}{\Lambda} & \frac{-J_{xz}^2 Q + J_y J_z Q - J_z^2 Q}{\Lambda} \\ \frac{-J_x R - 2J_{xz} P + J_z R}{J_y} & 0 & \frac{-J_x P + 2J_{xz} R + J_z P}{J_y} \\ \frac{J_x^2 Q - J_x J_y Q + J_{xz}^2 Q}{\Lambda} & \frac{J_x^2 P - J_x J_{xz} R - J_x J_y P + J_{xz}^2 P + J_{xz} J_y R - J_{xz} J_z R}{\Lambda} & \frac{-J_x J_{xz} Q + J_{xz} J_y Q - J_{xz} J_z Q}{\Lambda} \end{array} \tag{1}$$

Linearizing the system in SymPy, the root locus and Bode plots can be quickly computed in Python. These are shown in Figure 6 and 7 respectively for the transfer function relating the motor throttle command to the altitude of the quadrotor.

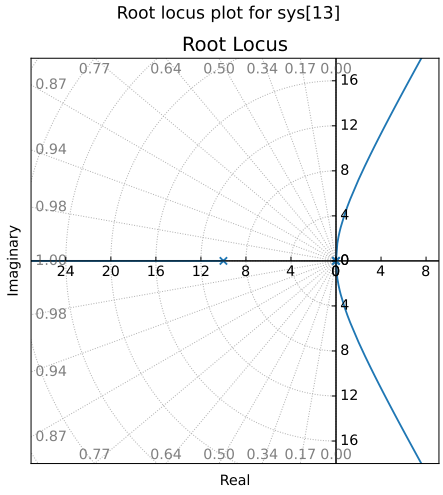

Figure 6. Root locus plot for linearized quadrotor.

5 Summary and Future Work

This paper introduces Rumoca, a Rust-based translator that automates the conversion of Modelica models into a variety of target algebraic modeling languages. Demonstrated through the translation of three models into two algebraic representations—CasADi and SymPy—Rumoca showcases its ability to bridge Modelica and common

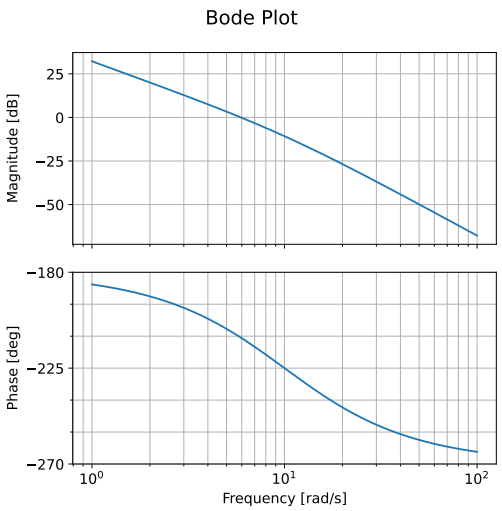

Figure 7. Bode plot for linearized quadrotor.

computational tools. While still in active development, future enhancements to Rumoca aim to expand its capabilities to handle a broader range of Modelica features. Additionally, future work may include the creation of Jinja templates to integrate with other back-ends, such as JAX for machine learning applications, Pycollimator for an intuitive graphical user interface, and Gazebo for simulations.

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.

We would like to thank Jörg Singer for his assistance extending the Parol parser generator to support our work.

References

(OSMC), Open Source Modelica Consortium (2025). *OpenModelica - Open Source Modelica Environment*. https://www.openmodelica.org. Accessed: 2025-04-30.

Agosta, Giovanni et al. (2023). "MARCO: An Experimental High-Performance Compiler for Large-Scale Modelica Models". In: *Proceedings of the 15th International Modelica Conference*. Modelica Association, pp. 1–10. DOI: 10.3384/ecp20413. URL: https://ecp.ep.liu.se/index.php/modelica/article/view/909.

Amrani, Moussa et al. (2021). "Multi-paradigm modelling for cyber–physical systems: A descriptive framework". In: *Software and Systems Modeling* 20.3, pp. 611–639. DOI: 10.1007/s10270-021-00876-z.

Andersson, Jakob 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/s12532-018-0141-4.

Andersson, Joel (2024-03). "Import and Export of Functional Mockup Units in CasADi". In: pp. 321–326. DOI: 10.3384/ecp204321.

Andersson, Joel and James Goppert (2025-01). "Event Support for Simulation and Sensitivity Analysis in CasADi for use with Modelica and FMI". In: pp. 99–108. DOI: 10.3384/ecp20799.

Biegler, Lorenz T. et al. (2009). *Large-Scale Nonlinear Optimization*. Vol. 83. Lecture Notes in Computational Science and Engineering. Springer. ISBN: 978-3-540-88321-3. DOI: 10.1007/978-3-540-88322-0. URL: https://doi.org/10.1007/978-3-540-88322-0.

Contributors, Pymoca (2025). *Pymoca: Modelica Compiler in Python*. https://github.com/pymoca/pymoca. Accessed: 2025-04-30.

Henriksson, Carl Johan et al. (2011). "Modelica: A unified object-oriented language for modeling and simulation of complex systems". In: *Simulation Modelling Practice and Theory* 19.3, pp. 697–718. DOI: 10.1016/j.simpat.2010.10.005.

Index, Python Package (2024). *Jinja2: PyPI Package Statistics*. https://pypistats.org/packages/jinja2.

Jung, Ralf et al. (2018). "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proceedings of the ACM on Programming Languages* 2.POPL, p. 66. DOI: 10.1145/3158154.

Klabnik, Steve and Carol Nichols (2019). *The Rust Programming Language*. No Starch Press. URL: https://doc.rust-lang.org/book/.

Liu, Bo et al. (2020). "A survey of model-driven techniques and tools for cyber-physical systems". In: *Frontiers of Information Technology & Electronic Engineering* 21.11, pp. 1567–1590. DOI: 10.1631/FITEE.2000108.

Matsakis, Nicholas D. and Felix S. Klock (2014). "The Rust Language". In: *ACM SIGAda Ada Letters* 34.3, pp. 103–104. DOI: 10.1145/2660193.2660199.

Meurer, Aaron et al. (2017). "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3, e103.

Modelica Association (2023). *Base Modelica Requirements — MCP-0031*. Accessed: 2025-07-24. URL: https://github.com/modelica/ModelicaSpecification/blob/MCP/0031/RationaleMCP/0031/Base-Modelica-requirements.md.

Projects, Pallets (2024). *Flask Documentation*. https://flask.palletsprojects.com/.

Red Hat, Inc. (2024). *Ansible Documentation*. https://docs.ansible.com/.

Singer, Jörg (2025). *Introduction to the Parol Parser Generator*. https://jsinger67.github.io/Introduction.html. Accessed: 2025-05-02.