
From MetaModelica to Rust

A working Rust compiler, a WebAssembly simulation target, and a proposal for the code base

AGENDA

What This Talk Covers

- 01 Where the MetaModelica compiler stands today
- 02 What Rust and WebAssembly are
- 03 The port, and what the testsuite says about it
- 04 Simulation in WebAssembly, with a live demo
- 05 Open concerns, migration plan, and the decision

PART ONE

Background

BACKGROUND

The MetaModelica Compiler Today

A language we designed

MetaModelica extends Modelica with algebraic data types, pattern matching and garbage collection, for writing compilers.

A compiler that compiles itself

The OpenModelica compiler is bootstrapped: building it requires a working copy of itself, which we also maintain.

Two decades of work

The frontend, backend and code generators in the repository are the product of a long line of contributors.

BACKGROUND

The Cost of Maintaining Our Own Language

Bootstrapping

A second compiler to build, test and release before any change ships.

A narrow contributor pool

New developers must learn a language used by one project before they can fix anything.

No standard tooling

No IDE support, debugger, profiler or package manager beyond what we write ourselves.

Everything is in-house

Memory management, data structures and runtime support are ours to fix, not a dependency's.

BACKGROUND

What Rust Is

A compiled systems language with no garbage collector, developed in the open and used in Linux, Firefox, Windows and Android.

Memory safety

Ownership rules are checked at compile time. No use-after-free, no data races.

Pattern matching

Algebraic data types and exhaustive match — the constructs MetaModelica was built to provide.

Cargo and the crate ecosystem

One command adds a dependency and it builds everywhere. Build system, formatter, linter, test runner, debugger and profiler ship with the language.

Interoperable in both directions

Crates bind to C, C++ and Julia, and can be published as Python modules, so migration can happen piece by piece.

BACKGROUND

What WebAssembly Is

A portable binary instruction format. Rust, C and C++ all compile to it.

Runs everywhere

In every current browser, and in standalone runtimes on the desktop and in CI.

Sandboxed by default

A module reaches only the memory and the imports it is given. Untrusted models stay contained.

Compiled, not interpreted

Runtimes JIT-compile modules to native code, which is why simulation speed stays close to native.

PART TWO

The Rust Port

MetaModelica and Rust Side by Side

MetaModelica

```
function simplify
  input Exp inExp;
  output Exp outExp;
algorithm
  outExp := match inExp
    local Exp e1, e2;
    case BINARY(e1, ADD(), e2)
      guard isZero(e2)
      then simplify(e1);
    case BINARY(e1, MUL(), e2)
      guard isOne(e2)
      then simplify(e1);
    else inExp;
  end match;
end simplify;
```

Rust

```
fn simplify(e: Exp) -> Exp {
  match e {
    Exp::Binary(l, Op::Add, r)
      if r.is_zero() => simplify(*l),
    Exp::Binary(l, Op::Mul, r)
      if r.is_one() => simplify(*l),
    other => other,
  }
}
```

Status of the Port

The MetaModelica to Rust compiler is finished and the translated compiler runs.

Testsuite results are identical to the results before the translation.

The existing C/C++ code generators are untouched and still work as before.

PART THREE

WebAssembly as a Simulation Target

Simulation Without a C Compiler

The JIT target compiles the flattened model straight to WebAssembly and executes it in-process.

No C toolchain for the user to install.

No C source generated and written to disk.

Compilation time drops from minutes to seconds.

Without code generation and linking, the testsuite runs two to three times faster on the same machines.

Measured speed exceeds the general figure because only the generated equations run as WebAssembly; the runtime stays compiled, and the generator uses more efficient data structures. Library testsuite, 8 h 19 m against 10 h 08 m total simulation time.

WebAssembly in general

≈85%

of native C speed

OpenModelica WASM-jit, measured

122%

of the C target's simulation speed

Coverage on the WASM JIT Branch

102

libraries tested on every
branch, the full
OpenModelica library set

18,255

of 20,005 models simulate
on the WASM JIT branch,
against 18,246 on master

10,186

models verify against
reference results, against
10,170 on master

26 h

to run the whole set,
against 50 h on master

LIVE DEMO

Modelica in the Browser

playground.openmodelica.org/latest/

build.openmodelica.org/Documentation/

No installation, no account, no server doing the work. The compiler and the simulation both run in the browser tab.

Open Concerns

Learning curve

Current developers have to learn Rust. Borrowing and lifetimes take time to absorb, and the first months will be slower.

Pattern matching ergonomics

Rust matching is close but not identical. Boxed recursive types and explicit clones make some compiler code more verbose.

Dependence on one language

We would tie the code base to Rust's long-term health, rather than to a language we control ourselves.

Migration Plan

Underway

The simulation runtime (solvers, driver) and some shared code has been rewritten in Rust. Rust is C-compatible, so this works alongside MetaModelica today.

After 1.28 or 1.29

Once the release branches off, remove the MetaModelica sources and the bootstrapping build from the repository. Pending board approval.

The cost of waiting

Until then, both implementations have to be kept working. The MetaModelica to Rust translation is slow, and while MetaModelica changes mostly carry over, external C code and similar changes have to be made in two places.

The Decision

We ask the consortium to approve removing the MetaModelica code and moving the compiler code base to Rust.