A Portable and Secure Package Format for Executable Simulation Modules based on WebAssembly

Moritz Allmaras*[1] Andrés Botero Halblaub[2] Harald Held[2] Tim Schenk[2]

[1]Siemens Energy Global GmbH & Co. KG, Germany, moritz.allmaras@siemens-energy.com
[2]Siemens AG, Germany, {andres.botero,harald.held,tim.schenk}@siemens.com

Abstract

We propose a new format (Digital Twin Assembly – *dtasm*) for self-contained executable co-simulation modules that is portable and sandboxed, yet offers performance close to native machine code and is sufficiently lightweight for running on embedded devices. *Dtasm* is based on WebAssembly, a standardized bytecode format for a stack-based virtual machine originally developed for high-performance computations in web browsers. A language-independent binary interface for such modules is described that is functionally comparable to FMI for co-simulation but not tied to a particular programming language. We discuss the benefits and drawbacks of this approach and how it can address some specific issues for executable simulation modules running in parallel with the operation of real systems.

Keywords: Simulation Modularization, Portability, Sandboxing, WebAssembly

1 Introduction

Recent industry trends towards digitization of production facilities, plants, infrastructure and transportation have amplified the need for digital twins not only during the design and engineering of such systems, but also for supporting their commissioning, automation and control during operation. Use cases for such digital twins performing online simulation range from virtual sensing, model predictive control and anomaly detection to optimal operation scheduling (Boschert, Heinrich, and Rosen 2018; Tao and Zhang 2017; Rasheed, San, and Kvamsdal 2020).

In contrast to the offline use of modeling and simulation during design and engineering of systems, the execution of numerical simulations in parallel to the operation of a real-world system presents some challenges that typically do not arise in offline scenarios:

- Computation needs to be sufficiently fast to keep up with the progress of the real-world system, so some kind of (soft or hard) real-time constraint needs to be fulfilled.
- The simulation needs to run robustly and reliably without human intervention for extended periods of time.
- Failure modes need to be predictable and their effects deterministic. Online simulations often perform safety critical tasks in control and automation systems for which rigorous regulations regarding testing and certification procedures are applicable.
- The hardware on which online simulations are executed is heterogeneous. Computing devices used for carrying out online numerical simulations often need to operate close to the "shop floor" of the actual physical systems to keep signal latency low. Hence, the hardware in use varies between different plants. The type of devices in use ranges from specialized microcontrollers (MCUs) to programmable logic controllers (PLCs) to industrial PCs depending on the specific application and scenario.

While the significance of each of these requirements varies from application to application, they are major contributors to the fact that online simulations in industrial applications are often customized solutions and cannot easily be re-used. Also, many established system simulation tools have their own ways of exporting online-capable simulations, often through code generation or compilation of binaries with a proprietary API (see, e.g., Schijndel (2014)), which further limits the reusability and composability of the resulting executable simulation modules.

On the other hand, as most of the industrial systems of interest are composed of smaller subsystems and components, the digital twin of such a system could also be modeled as a composition of smaller, independent subsystem or component twins. Just like for a physical asset such as a pump, gear box or conveyor belt, the same brand and model is deployed in many different real-world systems, the same should be possible for their digital twin counterparts: The digital twin of a component should be independent of the authoring tool used for its creation and be re-usable across as many different contexts and environments as possible.

In the future, components may be equipped with their own digital twins from the factory, and modularity is a major requirement for being able to integrate such supplier-provided twins into a complex system simulation. An implication of such a scenario is that the authors of digital twins are different from the plant operators running the

*Corresponding author

twins on their infrastructure. Hence, an elevated level of trust between producers and consumers of online digital twins is required, and security boundaries need to be defined that encapsulate supplier-provided twins in a safe, yet performant sandbox during their online execution.

In this text, we present a new format for self-contained executable digital twins that is both portable and sandboxed, yet allows close-to-native compute performance and is sufficiently lightweight to be used on embedded systems.

1.1 State of the Art

The modularization of industrial simulations has started to gain traction with the more widespread adoption of the *Functional Mock-Up Interface* (*FMI*) standard (Blochwitz, Otter, Arnold, et al. 2011; Blochwitz, Otter, Akesson, et al. 2012), which specifies a tool-independent interface and packaging format (*Functional Mock-Up Unit (FMU)*) for simulation modules. The *FMU* format allows system simulation tools to export self-contained simulation modules in such a way that they can be reused in different environments and by different tools than they have been authored in. FMI distinguishes *Model Exchange (ME)* and *Co-Simulation (CS)* modules, where Co-Simulation modules include a numerical solver and hence are most suitable for packaging executable simulations. FMUs may contain code as binaries ("binary FMU") or C sources ("source FMU") or both. The code contained in an FMU exposes a set of C functions that are specified by the FMI standard.

However, in the online simulation scenarios outlined above, FMI also presents some challenges regarding portability and the enforcement of security boundaries between the co-simulation master and the FMU instances:

- Binary FMUs only support the target platforms they have been explicitly compiled for, i.e. the relevant target platforms have to be known at compile time.
- Native binaries are difficult to sandbox from their executing environment. If loaded into a native process, an FMU assembly can directly interact with the OS kernel through system calls, and hence affect overall system integrity. Hence, the use of native binaries in-process requires a high level of trust in the authoring party.
- From the point of view of the embedding application, it is hard to determine upfront if a binary FMU is actually self-contained or requires additional dependencies to be dynamically linked at runtime (such as specific version of C or C++ runtime libraries). The availability of the correct version of such runtime dependencies has to be ensured though, and they are not specified in the model description.
- The runtime interface of FMI is specified in terms of C function calls, hence implementation of the interface in programming languages other than C and C++ need to rely on the *foreign function interface* (*FFI*) mechanisms of the respective programming language. While this is common practice in most programming languages, the implementation of the FMI runtime interface is often less ergonomic and safe than in C and C++.
- Source FMUs expose their internal implementation and thus are not viable in many industrial contexts where intellectual property (IP) protection is paramount.
- Source FMUs need an extra build step before they can be executed, and the FMI standard does not specify the details of this build step. Consequently, the build step is often proprietary to the generating tool and thus difficult to automate across FMUs created by different tools.

These limitations can impact the ability to exchange and re-use FMUs across different applications and hardware environments. In particular, additional measures are necessary to enforce security boundaries between the host environment and the code supplied by an FMU. An example for such measures is execution in separate processes connected through an interprocess communication (IPC) mechanism (see e.g. Hatledal et al. (2019)). However, the additional operational complexity of such multi-process setups is considerable, and on many embedded targets the necessary infrastructure and resources may not be available.

1.2 Digital Twin Assembly

Digital Twin Assembly is based on WebAssembly (Haas et al. 2017), a W3C-standardized bytecode format for a stack-based virtual machine, that has originally been developed to enable high-performance computations inside web browsers. The core WebAssembly specification (Rossberg 2019) is slim and low-level and is meant as a compilation target for compilers of high-level programming languages. Since it is independent of any other web technology, WebAssembly has recently seen increasing adoption in applications outside web browsers, such as server-side execution of user-supplied code (Hall and Ramachandran 2019), smart contract applications (Zheng et al. 2021) and Internet of Things (Jacobsson and Willén 2018). We define a simple and portable interface to such WebAssembly modules that functionally resembles FMI for co-simulation, but is not tied to specific platforms or language ecosystems.

1.3 Outline

Section 2 discusses the available options for the packaging of executable simulation modules. In section 3, WebAssembly as the target format of *dtasm* is introduced, as well as the application binary interface (ABI) that has been developed to interact with simulations packaged as WebAssembly modules, and the strengths and weaknesses of

DOI 10.3384/ecp21181443

the proposed format are discussed. Section 4 deals with prototypical implementations of *dtasm* runtimes and modules that were created in the course of this work. The performance of *dtasm* is compared with that of native binaries for some exemplary cases. In section 5, we summarize our findings and give an outlook on some further topics regarding online simulation that *dtasm* could potentially help to address.

2 Packaging of Executable Simulation Modules

In this text, the definition we will use for executable simulation modules is very similar to co-simulation modules in the sense of FMI:

- Modules carry a machine-readable description of the model containing information such as model metadata, input, state and output variables, default values, validity ranges for experimental conditions and capabilities of the module's implementation.
- Modules allow the creation of independent instances of the executable simulation.
- Values for constant parameters can be supplied and initial conditions for state variables can be set.
- Progressing the module instance's state from t_i to t_{i+1} consists of the steps:
  1. Values for the input variables at time t_i are supplied to the module instance,
  2. a time step from t_i to t_{i+1} is calculated,
  3. values for output variables and states at time t_{i+1} are returned.
- The internal state of a module instance can be reset to the time step immediately preceding the current time step.
- Instances can be terminated and disposed of at any time.

The normal sequence of invocation for online simulations (without considering potential reset of timesteps) is depicted in 1. Just like for a co-simulation FMU, no restrictions on the internal implementation of the simulator are imposed. It could, e.g., implement a numerical solver for a differential algebraic equation (DAE), a forward evaluation of a trained machine learning model or even some simple table lookup mechanism.

2.1 Packaging Options

For packaging such executable simulation modules, there are two common variants:

1. Packaging native machine code targeting certain platforms.
2. Packaging the simulation's source code in some given programming language.

Binary packaging allows only the explicitly supported platforms to execute the simulation modules. On other platforms, virtualization mechanisms could be utilized, but in practice such virtualization is complex and expensive in terms of the needed compute and memory resources and hence often not a feasible option at least on embedded devices.

For source packaging, the sources need to be compiled to native machine code by the embedder of the module prior to execution on the target hardware. This allows the source code to be compiled by specialized compilers for the target hardware. In this case, the packaging format needs to specify the exact supported feature set of the programming language, as well as all necessary operations for compiling the source code to native machine code. This option places the burden of compilation on the embedder of the module, which in many cases necessitates manual intervention and prevents the automated deployment of such packages. Providing simulation's source code also exposes the intellectual property of the implementation, which is frequently a major obstacle for the adoption of such package formats in industrial contexts.

Another option somewhat in between 1 and 2 is the packaging of intermediate bytecode targeting a virtual instruction set architecture (ISA). For execution, the bytecode is then either interpreted by an application-level virtual machine or compiled to native machine code prior to execution. Well-known examples of such bytecode formats include Java bytecode, Common Intermediate Language (CIL) (as used by the Common Language Runtime of the .NET platform) and Python bytecode (used by CPython). Traditionally, bytecode formats have not received much attention as a target format for numerical computations since they are considered slow in comparison to native machine code due to the overhead incurred by interpretation or compilation. However, we believe that bytecode has some considerable advantages especially when used in online scenarios. Bytecode formats are very portable since they do not depend on a certain hardware instruction set, and they allow efficient sandboxing of executable code by limiting access to resources and intercepting system calls. However, many of the existing bytecode formats are rather complex and have explicit support for some of the high-level constructs of the corresponding ecosystem (like garbage collection). Hence, many of the existing bytecode formats are a poor fit as compile targets for system-level programming languages (like C, C++ or Fortran) that are commonly used in numerical simulation.

3 Digital Twin Assembly Format

3.1 WebAssembly Bytecode

Starting in 2015, a bytecode format for a stack-based virtual machine called WebAssembly (Wasm) has been developed by a working group of the World Wide Web Consortium (W3C). Since then, WebAssembly has reached stable version 1.0 and gained the status of a W3C-recommended standard (Rossberg 2019). Its original goal is the high-performance execution of computational logic

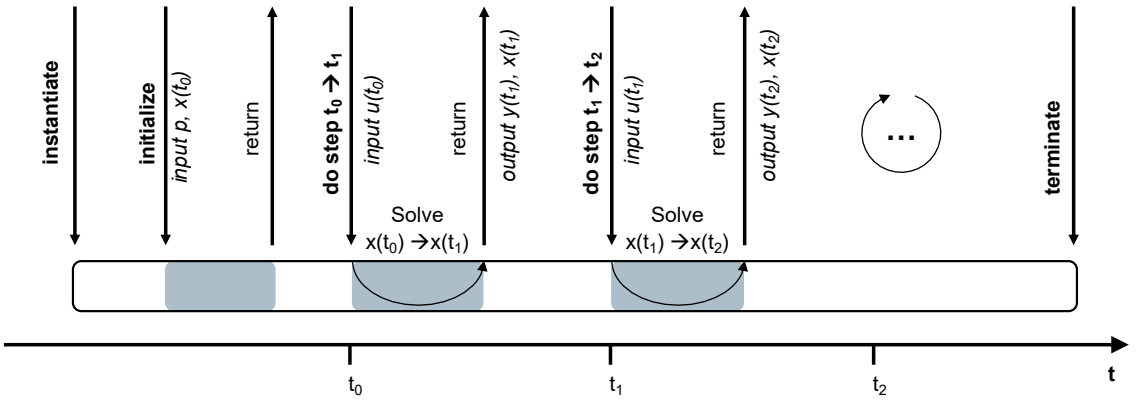

Figure 1. Lifecycle and call structure for executable simulation modules.

in web browsers. The WebAssembly specification defines a narrow low-level instruction set, with the intention that support for emitting Wasm bytecode can be easily added to existing compiler toolchains. Most notably, the LLVM compiler infrastructure (Lattner and Adve 2004) was among the first to include a backend for generating Wasm output, such that any programming language for which an LLVM frontend exists can be compiled to Wasm bytecode. In addition, WebAssembly is strongly and statically typed, and has a deterministic stack behavior that can be statically analyzed, allowing interpreters and compilers to aggressively optimize execution of the code. Consequently, WebAssembly modules can be executed with close to native performance in many scenarios (Jangda et al. 2019). The narrow instruction set also allows the implementation of lightweight interpreters for execution on small, resource-constrained devices (Peach et al. 2020). Sandboxing of the bytecode execution from the host environment has been an explicit design goal of the WebAssembly specification, since it is a paramount requirement for use in web browsers, where individual browser windows need to be kept isolated from each other and the host environment. Wasm modules cannot directly access host memory or invoke system calls. Instead, memory is provided as a contiguous linear block, and access to this block is bounds-checked by the WebAssembly runtime. System calls or calls to external libraries need to be explicitly enabled by the runtime (opt-in model) in order to be callable from inside the sandbox. WebAssembly modules are statically linked and do not (yet) support dynamic linking, so other than function imports and exports, they are self-contained. In light of the requirements for online digital twins discussed in section 1, WebAssembly offers some unique advantages over binary and source packaging:

- The bytecode is portable and can be executed on any hardware for which a Wasm runtime exists.
- Performance can be close to that of native machine code, at least in environments where just-in-time (JIT) or ahead-of-time (AOT) compilation to native machine instructions is possible.
- Module instances are sandboxed and cannot interfere with each other or the host environment in uncontrolled ways.
- Implementation of modules can be carried out in any programming language that supports compilation to Wasm.
- Module code can be statically analyzed for memory usage and instruction counts.
- The runtime has complete control over the execution such that running bytecode instances can be preempted and a resumable snapshot of an instance's state can be taken by the runtime without requiring explicit support by the module implementation.
- There is no undefined behavior, all operations are deterministic and hence the computed outputs are identical across different Wasm runtimes and host environments.

As downsides of this approach the overhead due to the WebAssembly runtime needs to be mentioned (unless the modules are AOT compiled, which is only possible on limited set of platforms, and adds an additional compilation step before execution). On platforms where JIT or AOT compilation is not available (e.g., on many embedded devices), Wasm modules need to be interpreted which causes substantial performance degradation.

A WebAssembly module may interact with its host environment through imported and exported functions. Function imports are declared by name and signature and linked by the runtime when the module is instantiated. Function exports are also declared by name and signature and can be called by the runtime once the module is instantiated. A further mechanism for exchanging data is the use of the linear memory blocks. A schematic overview of the interactions between a Wasm module and its host is shown in 2.

3.2 Interface

According to the description of executable simulation modules given in Section 1, the interface that the module

DOI 10.3384/ecp21181443

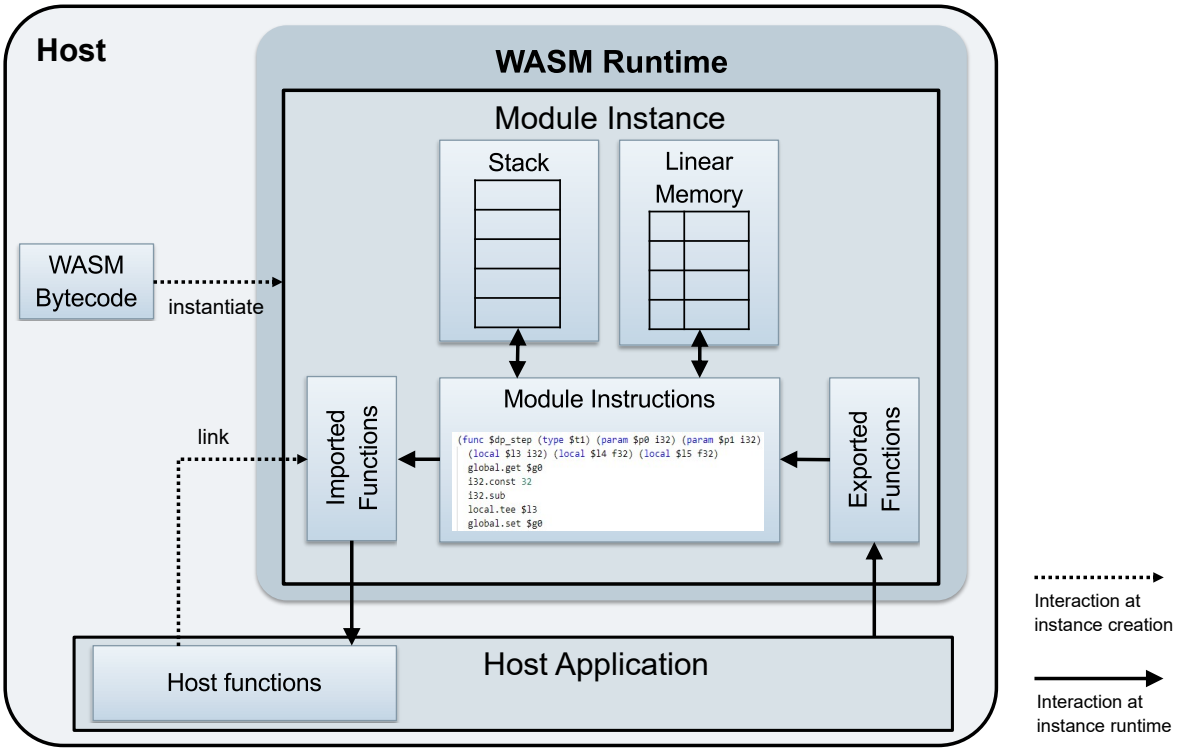

Figure 2. Interactions between WebAssembly modules and the host.

exposes includes the following functionality:

- Retrieve a model description from the module specifying input, state and output variables, module capabilities as well as potential constraints on compatible timestep lengths,
- create a new instance of the simulation module and initialize it with given parameters and initial values,
- set values for the input variables,
- calculate forward in time by a given timestep,
- retrieve the resulting values of the output and state variables,
- reset state to the previous timestep,
- terminate the instance.

The creation of such high-level interfaces between WebAssembly modules and their host is complicated by the fact that core WebAssembly only knows four basic data types (32bit and 64bit variants of integers and floating point numbers), and the signatures of any declared export functions need to be expressed in terms of these data types. More complex data structures can only be exchanged by serializing them to linear memory, which is accessible both to the module instance and to the host. Then, pointers to locations inside linear memory can be passed as arguments to a regular Wasm exported functions (pointers into linear memory are just offsets from the start of the memory block). To handle heap allocation inside the linear memory in a consistent way, a *dtasm* module exports an allocator and a corresponding de-allocator function.

Since serialization and de-serialization need to be performed on either side of the Wasm sandbox for this to work, a serialization format should be chosen that is not just lightweight, but also has implementations in many different programming languages. After careful consideration, we picked the FlatBuffers ((FlatBuffers 2021)) serialization library for the *dtasm* ABI. In FlatBuffers, data structures are described by a schema written in an interface definition language (IDL), and a compiler provided by the FlatBuffers project then generates source code for serialization/de-serialization of such structures for a variety of target languages. The generated code is very performant and lightweight (e.g., the code generated for C++ is a single, self-contained header file), and code generation supports a wide range of contemporary programming languages. Furthermore, FlatBuffers (at least in some programming languages) allows the validation of binary buffers for a given schema. This is an important feature for enforcing the security boundary between host and modules and for increasing the robustness of implementations.

The sequence of events for invoking a *dtasm* interface function generally follows these steps:

- The host assembles the input data into a FlatBuffer, determines its size and invokes the allocator function exported by the *dtasm* module instance to allocate a buffer in linear memory of the instance.
- The host serializes the input FlatBuffer into the allocated memory block.
- The host allocates an additional buffer (of a default size) for holding the results of the call.
- The host invokes the interface function, passing pointers to the in- and output buffers as well as their

respective sizes.
- The module decodes the input buffer, processes it and assembles the result into a result FlatBuffer.
- If the output buffer is sufficiently large to hold the result FlatBuffer, the result is written to the output buffer and the call returns. If not, the size of the result is returned, the host allocates a new large enough output buffer and invokes the interface function again with the same input.
- The host reads the result FlatBuffer from the output buffer.

For a given programming language, much of the logic involved in this procedure can be encapsulated into auxiliary libraries, such that the consumers of the ABI don't need to deal with the low-level details.

3.3 Model Description

Similar to the model description defined by the FMI standard, a *dtasm* module provides a model description that contains

- metadata about the module (name, id, when and by what tool it was created),
- module capabilities (e.g., can timesteps be reset, can the module utilize derivative information),
- a list of model variables together with their causality, data type and default value,
- infos about valid experiment conditions such as constraints on compatible timestep size, start and end time of simulations.

Model variables can be of causality parameter, input, local or output, and the supported data types are real, integer, boolean and string.

A more detailed description can be found as part of the code repository in (dtasm 2021). The binary format of the model description is again described by a FlatBuffers schema. Since FlatBuffers supports creation of buffers from JSON files that are compatible with the schema, model descriptions can be authored using JSON for convenience. The binary representation is then embedded into the module as a byte array literal, and can be retrieved from instances of the module by invoking the `getModelDescription` interface function.

Since module instantiation is already a part of the WebAssembly specification, no explicit interface function is needed for instantiation. Likewise, since WebAssembly modules cannot use any native resources, an explicit `terminate` function in the interface is not needed and instances can be terminated and disposed of simply by unloading them from the WebAssembly runtime.

4 Features and Limitations of *dtasm*

4.1 Features

Using WebAssembly bytecode as the target format for executable simulation modules has some interesting implications that we discuss in the following. As WebAssembly is a very simple bytecode format, it is easy to target by compilers for high-level programming languages, which is confirmed by the number of existing compilers supporting Wasm as output. On first look bytecode seems like an unusual format for executable numerical code. But considering, e.g., the LLVM compiler architecture (Lattner and Adve 2004), it is based on a separation between frontend and backend compilation, where the frontend generates intermediate bytecode (LLVM Intermediate Representation (IR)) that is compiled to native machine code by the backend. WebAssembly can be thought of as replacing the intermediate bytecode by a portable, well-specified format that can be easily targeted by other compilers as well. The development of tools, infrastructure and supporting standards around WebAssembly has been strongly driven by the Web community during recent years, which has lead to a number of high-quality implementations and standards being available as open source (e.g. Zakai (2011), WASI (2021), and AssemblyScript (2021)).

Instances of WebAssembly modules can store internal state on the stack, in linear memory or in global variables (but as globals are seldomly used for this purpose, we disregard them here). When a module instance is not currently executing a function, its stack is empty, so that a snapshot of its state can be created simply by dumping the content of its linear memory (which is just a contiguous byte array) to a file. The instance can then be terminated, a new instance be created and its linear memory read back from the file, and the new instance then has exactly the same internal state as the previous one. All this can be achieved solely from the runtime without any explicit support by the module implementation. The memory dump is even portable across different WebAssembly runtimes, as the mechanism of linear memory is specified by the WebAssembly standard. E.g., this method could also be used to reset timesteps for modules that do not explicitly support such functionality:

1. Store a dump of the linear memory after each timestep.
2. If a timestep needs to be reset, the previous dump is loaded into the instance's memory to reset its state.

Depending on the size of the module's memory, this procedure can be quite expensive, hence explicit timestep resetting support by the module should be preferred when available.

Some more advanced features could even include preemption of running module instances by the runtime, relocation to other machines and resumption at the exact state where preemption happened. Such operations are not yet widely supported by popular Wasm runtimes, but many projects are rapidly adding features in this direction. Preemptive multitasking could prevent individual module instances from occupying computational resources and allow a fair distribution of resources to all running instances. Related is the concept of gas counting: The runtime can monitor the consumed instruction count ("gas") of a We-

DOI 10.3384/ecp21181443

bAssembly function and preempt the instance if the instruction count exceeds a certain threshold. This could allow a fair distribution of compute resources among multiple active module instances.

4.2 Limitations

The WebAssembly standard in its current stage has some limitations that impact its usefulness as a packaging format for executable simulations:

- Not all features of low-level programming languages can be mapped cleanly to WebAssembly. In particular, non-local jumps, stack unwinding or multithreading do not currently have support in WebAssembly, although extensions of the standard for supporting these features are planned.
- Specialized hardware acceleration units like GPUs or TPUs are not accessible to WebAssembly modules. Support would need customized implementations outside of the Wasm specification.
- WebAssembly modules are statically linked, which makes them rather large in size (an extension of the Wasm spec allowing dynamic linking is planned).
- Available development tooling, especially in regards to debugging support, is lacking behind other more established ecosystems.
- The size of linear memory blocks is given in multiples of 64kB, which is wasteful on embedded platforms.
- Security of the Wasm sandbox model is not perfect, e.g., side-channel attacks are not prevented by the specification but need to be mitigated by runtime individually.

Also, the general overhead of a WebAssembly runtime in terms of performance and memory usage is certainly not negligible. Especially on embedded platforms, AOT compilation or JIT compilation are often not available or not feasible, so the only option are interpreters that are generally an order of magnitude slower than native code (see Wasm3 (2021)). Very small devices with less than 64kB memory or no support for dynamic memory allocation are not suitable for running *dtasm* modules. Performance and size of the runtime is often a tradeoff: While interpreters can be very lightweight (Wasm3 is around 100kB in size when compiled), JIT runtimes on the other hand include native code generators and thus are often several tens of megabytes in size.

5 Prototypical Implementation

Several implementations of *dtasm* runtimes and modules have been developed during the course of this work, some of which are available as open source (dtasm 2021).

5.1 Runtimes

Dtasmtime is a *dtasm* runtime library implemented in Rust that builds upon (Wasmtime 2021), a popular open source engine for WebAssembly modules featuring JIT compilation. *Dtasmtime* supports loading and execution of *dtasm* modules as well as saving and loading of instance state to and from files. Interaction with *dtasmtime* from Rust applications happens through a high-level API, while an additional lower-level C-compatible API is provided in order to facilitate integration of the library into C/C++ and other programming languages.

Additional *dtasm* runtimes have been implemented based on the Wasm3 interpreter (Wasm3 2021) and the V8 JavaScript engine. While Wasm3 by nature of interpretation is substantially slower in execution performance than JIT or AOT compiling runtimes, it is very lightweight and allows execution of *dtasm* modules on embedded targets such as Arduino-class microcontrollers (see Figure 3). Implementation of a *dtasm* runtime in JavaScript allowed running *dtasm* modules inside contemporary web browsers as well.

5.2 Modules

For demonstration and benchmark purposes, a simple double pendulum simulator (based on Wheatland (2004)) has been implemented in C++ and Rust, and compiled into a *dtasm* module using the WASI SDK (2021) in the case of C/C++ and Rust's integrated `wasm32-wasi` target. Source code for both versions is available (dtasm 2021).

During the course of our experimentation, we also compiled *dtasm* modules from several source FMUs created by various commercial and open source simulation tools (Simulink/FMIKit, Dymola, OpenModelica). While most of the resulting *dtasm* modules could be successfully compiled and executed, some FMUs were found to utilize C/C++ functionality for error handling (e.g., exceptions, non-local jumps) that are currently not supported by WebAssembly and had to be stubbed in order to compile successfully. Such *dtasm* modules were then only operable under non-error conditions. As some FMUs utilized resource files that are read at runtime (which core WebAssembly does not support), the WASI interface for reading files from the host file system had to be made available when executing such *dtasm* modules. Accessing external files violates the self-containedness assumption on *dtasm* modules and also may have security implications. If a direct *dtasm* export was integrated into such simulation tools (without the detour through FMU), this issue could be avoided by embedding additional resource files directly into the WebAssembly module.

5.3 Performance

One of the most interesting questions regarding *dtasm* is the resulting performance overhead when comparing to execution of native machine code, since this is one of the major tradeoffs incurred by *dtasm*. This overhead consists of several distinct contributions:

1. Raw computational performance of WebAssembly compared to native machine code,
2. overhead afforded by the module interface, mainly through copying of memory blocks and

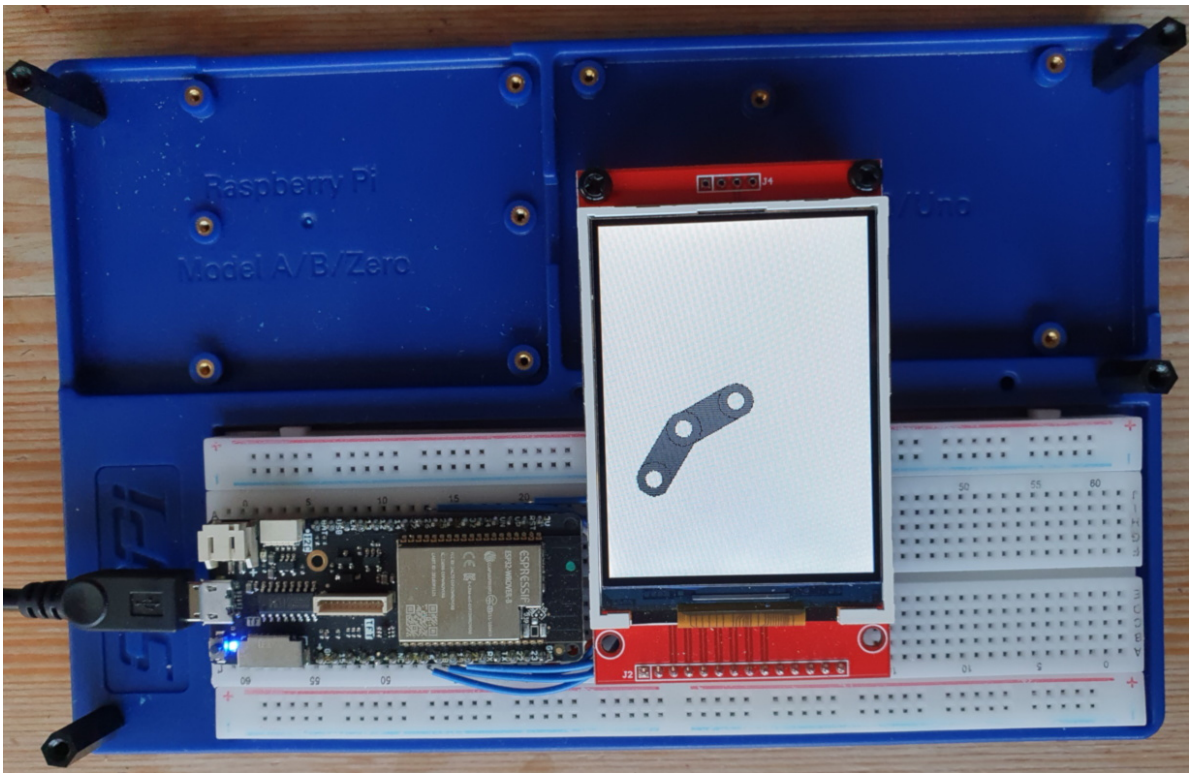

Figure 3. Double pendulum simulator generated from a Simulink model running as *dtasm* module on an ESP32 MCU.

serialization/de-serialization,

3. performance overhead due to how efficiently the Wasm runtime implements calls to Wasm export functions and access to the Wasm linear memory,
4. optimization capabilities of the compiler used to create the native machine code and the Wasm module respectively.

Characterization of these overheads in isolation is difficult and will not be attempted here. The raw performance overhead of Wasm for different engine implementations has been the subject of many benchmarks (see, e.g., Jangda et al. (2019), Denis (2021), and Wasm3 (2021)), in which a best-case factor for JIT-based engines between 1.5 and 2.5 has been found, depending on the workload and Wasm engine considered.

To compare performance of the *dtasm* prototype implementation to native execution, we used the C++ source code of our double pendulum module and added an outer loop that runs the simulation for a fixed number of steps, still using the *dtasm* interface but directly from C++. This combination was then compiled to native machine code using GNU compiler collection (gcc). We performed the same computation using the LLVM-compiled *dtasm* module running in *dtasmtime*, and compared execution times. Figure 4 shows the result for 10 million time steps of the double pendulum simulator. The overhead of *dtasmtime* is found to be around a factor of 2.4.

In an attempt to reduce the influence of 2 and 3 above (in particular the overhead incurred by the *dtasm* interface), we adapt the implementation of the double pendulum simulator to internally perform many small time steps of fixed size, and reduce the number of steps on the outer loop, thereby invoking the *dtasm* interface less often than in the first case. Figure 5 shows the results for 10^5 inner time steps and 10^4 outer loop steps (amounting to 10^9 time steps total).

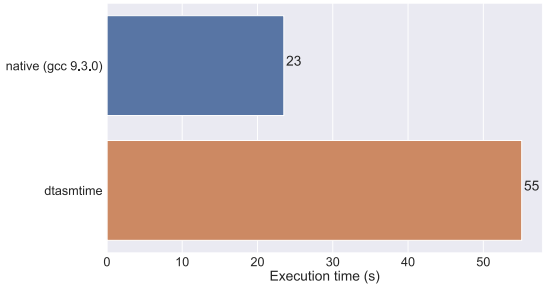

Figure 4. Execution times of the double pendulum simulator for 10^7 time steps.

It can be seen that native and *dtasmtime* performance are almost identical in this case. This implies that most of the overhead incurred by running simulations as *dtasm* modules indeed is due to interface calls. We note that the significance of this comparison is very limited though, because we only tested a single simulation module. Many features of more realistic simulators, such as extensive numerical linear algebra operations, may yield a different picture here. Also, no significant code optimizations have been applied to either the simulation module or the

DOI 10.3384/ecp21181443

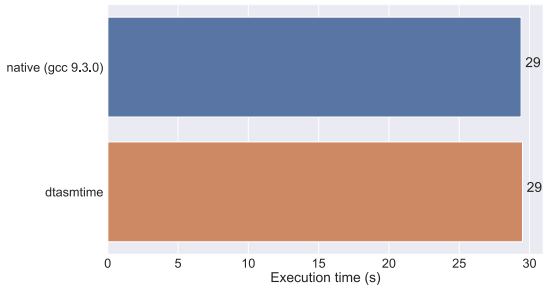

Figure 5. Execution times of the double pendulum simulator for 10^5 inner time steps and 10^4 outer time steps.

dtasmtime implementation.

The performance of Wasm interpreters such as Wasm3 was found to be around a factor of 10 slower compared to *dtasmtime*. Hence, the performance on embedded devices, where JIT compilation is not an option due to resource constraints, must be expected much lower than what is reported above. While AOT compilation to native code may be another option for such targets, it hinges on the availability of suitable compilers.

6 Conclusion

Packaging executable simulation modules as native machine code poses several challenges related to portability and security: Machine code targets specific hardware platforms and is difficult to sandbox from its execution environment. Bytecode formats can help address both of these issues since they target abstract machines with enforceable security boundaries. Bytecode formats can serve as compilation targets for higher level programming languages, and application level virtual machines for bytecode often support secure sandboxes by design. WebAssembly in particular is suitable for executable simulation modules as it focuses on performance and is sufficiently low-level to be used as compilation target for many of the programming languages typically in use by numerical codes.

In this text, we introduced an efficient and language-independent interface to WebAssembly modules that in functionality resembles FMI for co-simulation. We discussed the rationale for our design decisions as well as the advantages and drawbacks they entail. The design is sufficiently lean to allow targeting embedded devices, although the overhead created by the need for a virtual machine is certainly considerable there.

We demonstrated feasibility by providing prototypical implementations of *dtasm* runtimes and modules. A preliminary performance test shows that the main overhead is due to the module interface (which is not specific to WebAssembly), but the performance of Wasm itself can be expected comparable to the performance of native binaries. Simulation code generated by established system simulation tools can often be compiled into *dtasm* modules with manageable effort, allowing *dtasm* to take advantage of the existing system simulation ecosystem, e.g., through the export of source FMUs that are then compiled into *dtasm* modules.

Dtasm modules can be instrumented at runtime in a way that allows dynamic re-allocation to other compute nodes at runtime. In the future, this could enable orchestration systems that dynamically dispatch running module instances to compute nodes according to available resources. Compute nodes close to the shop floor could then be utilized as a single cluster instead of individually configured devices.

While WebAssembly is still a comparably young technology, it has beneficial properties regarding portability as well as sandboxing and shows promising results regarding performance. It remains to be seen if WebAssembly can be a relevant technology for packaging numerical simulations in the future. A further adoption would certainly hinge on support by existing system simulation tools to export *dtasm* modules. Using source FMUs as an intermediary for compiling to WebAssembly could be a viable path forward in this direction.

References

AssemblyScript (2021). “A language made for WebAssembly”. URL: https://www.assemblyscript.org/ (visited on 2021-03-14).

Blochwitz, Torsten, Martin Otter, J. Akesson, et al. (2012). “Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models”. In: *9th International Modelica Conference*. URL: https://elib.dlr.de/78486/.

Blochwitz, Torsten, Martin Otter, Martin Arnold, et al. (2011-03). “The Functional Mockup Interface for Tool independent Exchange of Simulation Models”. In: *8th International Modelica Conference*. Ed. by Christoph Clauß. Linköping Electronic Conference Proceedings. Linköping University Press, pp. 105–114. URL: https://elib.dlr.de/74668/.

Boschert, Stefan, Christoph Heinrich, and Roland Rosen (2018). “Next generation digital twin”. In: *Proc. TMCE 2018*. Vol. 2018, pp. 7–11.

Denis, Frank (2021). “Benchmark of WebAssembly runtimes – 2021 Q1 edition”. URL: https://github.com/jedisct1/webassembly-benchmarks/tree/master/2021-Q1 (visited on 2021-03-13).

dtasm (2021). “Digital Twin Assembly - A portable and sandboxed package format for executable simulation modules based on WebAssembly”. URL: https://github.com/siemens/dtasm (visited on 2021-04-29).

FlatBuffers (2021). “An efficient cross platform serialization library”. URL: https://google.github.io/flatbuffers/ (visited on 2021-03-13).

Haas, Andreas et al. (2017). “Bringing the Web up to Speed with WebAssembly”. In: *SIGPLAN Not.* 52.6, pp. 185–200. DOI: 10.1145/3140587.3062363.

Hall, Adam and Umakishore Ramachandran (2019). “An execution model for serverless functions at the edge”. In: *Proceedings of the International Conference on Internet of Things Design and Implementation*, pp. 225–236.

Hatledal, Lars Ivar et al. (2019). “Fmu-proxy: A framework for distributed access to functional mock-up units”. In: *Pro-*

ceedings of the 13th International Modelica Conference. Linköping University Electronic Press.

Jacobsson, Martin and Jonas Willén (2018). "Virtual machine execution for wearables based on webassembly". In: *EAI International Conference on Body Area Networks*. Springer, pp. 381–389.

Jangda, Abhinav et al. (2019). "Not so fast: Analyzing the performance of webassembly vs. native code". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 107–120.

Lattner, Chris and Vikram Adve (2004). "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, pp. 75–86.

Peach, G. et al. (2020). "eWASM: Practical Software Fault Isolation for Reliable Embedded Devices". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11, pp. 3492–3505. DOI: 10.1109/TCAD.2020.3012647.

Rasheed, Adil, Omer San, and Trond Kvamsdal (2020). "Digital Twin: Values, Challenges and Enablers From a Modeling Perspective". In: *IEEE Access* 8, pp. 21980–22012.

Rossberg, Andreas (2019). "WebAssembly Core Specification". URL: https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/ (visited on 2021-03-14).

Schijndel, A.W.M. (Jos) van (2014). "A review of the application of SimuLink S-functions to multi domain modelling and building simulation". In: *Journal of Building Performance Simulation* 7.3, pp. 165–178. DOI: 10.1080/19401493.2013.804122.

Tao, Fei and Meng Zhang (2017). "Digital Twin Shop-Floor: A New Shop-Floor Paradigm Towards Smart Manufacturing". In: *IEEE Access* 5, pp. 20418–20427.

WASI (2021). "The WebAssembly System Interface". URL: https://wasi.dev/ (visited on 2021-03-14).

WASI SDK (2021). "WASI-enabled WebAssembly C/C++ toolchain". URL: https://github.com/WebAssembly/wasi-sdk (visited on 2021-03-14).

Wasm3 (2021). "Performance". URL: https://github.com/wasm3/wasm3/blob/master/docs/Performance.md (visited on 2021-03-13).

Wasmtime (2021). "A small and efficient runtime for WebAssembly & WASI". URL: https://wasmtime.dev/ (visited on 2021-03-13).

Wheatland, Michael S. (2004). "The Double Pendulum". URL: http://www.physics.usyd.edu.au/~wheat/dpend_html/ (visited on 2021-03-14).

Zakai, Alon (2011). "Emscripten: an LLVM-to-JavaScript compiler". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 301–312.

Zheng, Gavin et al. (2021). "WebAssembly (WASM)". In: *Ethereum Smart Contract Development in Solidity*. Singapore: Springer, pp. 317–334. DOI: 10.1007/978-981-15-6218-1_11.