MODEL-DRIVEN ENGINEERING FOR HIGH-PERFORMANCE PARALLEL DISCRETE EVENT SIMULATIONS ON HETEROGENEOUS ARCHITECTURES

Romolo Marotta[1], and Alessandro Pellegrini[1]

[1]Dept. of Civil Eng. and Computer Science Eng., Tor Vergata University of Rome, Rome, ITALY

ABSTRACT

Modern high-performance, large-scale simulations require significant computational power, memory, and storage, making heterogeneous architectures an attractive option. The presence of accelerators in heterogeneous architectures makes model development hard. Domain-specific languages (DSLs) have successfully simplified model development, but designing a DSL to target heterogeneous architectures can be burdensome. Model-driven engineering (MDE) can simplify the development of DSLs targeting heterogeneous architectures. In this paper, we focus on MDE and propose a model-driven approach targeting Parallel Discrete Event Simulations on heterogeneous architectures. We exercise our MDE-generated models using a state-of-the-art runtime environment for heterogeneous architectures.

1 INTRODUCTION

Heterogeneous architectures are based on different types of processors, memory systems, and interconnects within a single computing system, or they employ different computing nodes that have different capabilities and/or different ways of executing instructions (Zahran 2017). They have become increasingly common in recent years, as the performance gains from traditional homogeneous architectures have slowed down, and have emerged as a promising approach for high-performance computing around the world (Gagliardi et al. 2019; Kothe et al. 2019).

Modern high-performance, large-scale simulations may require non-negligible computational power, memory, and storage, making heterogeneous architectures attractive. One of their main advantages is their ability to exploit the strengths of different types of processors. CPUs are well-suited for tasks that require complex control flow, while GPUs excel at data-parallel computations. FPGAs show a significantly reduced energy footprint. By combining these processors, heterogeneous architectures can achieve better performance and energy consumption than either type of processor alone. This is particularly important for large-scale simulations, which often involve a mix of computation and communication. Simulations in fluid dynamics, materials science, and astrophysics can require billions of computational elements, making them extremely demanding in terms of computational resources (see, e.g., Jameson and Vassberg 2001; Taffoni et al. 2019). As shown by Montesano et al. (2022), the workload mixture can be demanding for traditional CPU-only simulation environments. Hybrid architectures can provide the necessary level of performance and scalability needed for these simulations.

At the same time, heterogeneous architectures have introduced a new set of challenges in high-performance computing. Unlike traditional homogeneous architectures, heterogeneous architectures cannot hide their internal complexity from the programmer (Sutter 2014). This was already visible in multicore systems, where the multi-cache hierarchy required memory barriers to ensure the accuracy of algorithms. In heterogeneous systems, the problem is even more complicated because we need to move data around, in addition to synchronizing execution explicitly. The presence of accelerators in heterogeneous architectures makes programming even more challenging. These accelerators are optimized for specific calculations and require specialized programming models to exploit their potential fully. Programmers must, therefore, deal with the complexity of the underlying hardware and the intricacies of the programming models required

to use the accelerators. This is a significant departure from the traditional programming approach, where the hardware was completely abstracted away.

Programming a heterogeneous system can be particularly challenging as it involves multiple programming models and languages. For instance, when programming a system that includes CPUs, GPUs and FPGAs, we may need to use OpenMP or MPI, alongside CUDA or OpenCL for the GPUs, and HDL for the FPGAs. This results in a complex programming environment that can be difficult to manage and prone to errors. To fully exploit the potential of heterogeneous architectures, simulation models must be designed to take advantage of the different types of hardware available. This may involve partitioning the model into different components, each optimized for a specific type of hardware. For example, part of the model may be run on one accelerator while another part is run on a different piece of hardware, at the same time.

Moreover, the workload dynamics may change over time, requiring the re-orchestration of the model's execution. This may be necessary for performance reasons to ensure that the most compute-intensive parts of the model are running on the most appropriate hardware but it requires that almost the entire model's code is available in variants that can be run on either hardware device. Alternatively, it may be necessary for energy efficiency reasons to ensure that the model is running on the most power-efficient hardware available. As shown by Marotta et al. (2024), re-orchestrating the execution of simulation models on heterogeneous architectures is a complex task that requires sophisticated resource management techniques. These techniques must be able to monitor the workload dynamics and make real-time decisions about allocating resources to different parts of the model. This requires a deep understanding of the underlying hardware and the model's performance characteristics.

At the same time, simulationists are professionals who specialize in using computer simulations to study complex systems. They may not necessarily be computer scientists and are not concerned with the technical details of the underlying hardware. Instead, their focus is on understanding the physical phenomena and behavior of the systems being studied. To achieve this, they rely on computer simulations as a tool to gain a deeper understanding of these systems and make predictions about their behavior. As the models become more complex and the problems they are trying to solve become larger, the computational demands increase, requiring significant computer power. In such scenarios, heterogeneous supercomputers come into play. Therefore, the relevant question is how to identify solutions that enable domain experts to focus on the definition of the model or the problem they want to solve while hiding away the complexity of heterogeneous architectures to let them make their large-scale problems tractable.

Blunk and Fischer (2013), Warnke (2020) have shown that Domain-Specific Languages (DSLs) can be used to handle the increasing complexity of simulation models by tailoring software languages to specific application domains. This approach allows for clearer and more efficient expression of simulation models and experiments, as demonstrated by various examples in the field. While the DSL approach can solve the semantic need of simulationists, from an engineering perspective, they do not necessarily cope with the underlying hardware complexity. If different hardware is the target of the runtime support for one specific DSL, the compiling infrastructure must be adapted to deal with the idiosyncratic details of that specific piece of hardware (Uhrmacher et al. 2024). All this effort is demanded from the designers and developers of the DSLs. Multiply it for the large number of DSLs the simulation community may want to embrace, and the complexity of bridging the gap between simulationists and exascale-like supercomputers becomes too much.

A more viable approach we envisage in this paper is to rely on Model-Driven Engineering (MDE). MDE is an approach to software development that focuses on using models to design and build systems (Rodrigues da Silva 2015). In MDE, a *model* is an abstract representation of a system or its components, while *metamodels* define the structure and behavior of models and provide a common language for describing models across different domains. In the context of simulation, MDE and metamodels can help address the issues of designing and developing DSLs. Uhrmacher et al. (2024) noted that they can also improve non-functional properties of the simulation lifecycle (Uhrmacher et al. 2024). By defining a common metamodel for simulation models, we can provide a uniform representation that can be used across different simulation

tools and domains. This enables simulationists to write their models in a semantically-rich language while the compiling/runtime infrastructure takes care of the mapping of the model on the available hardware in a more simplified way.

MDE and metamodels can also help reduce the complexity of simulation modeling by providing a higher-level abstraction that hides the details of the underlying hardware and software. This can make it easier for simulationists to focus on the high-level design of their models rather than getting bogged down in the implementation details.

The rest of this paper is structured as follows. We discuss related work in Section 2. We identify and discuss the sources of heterogeneity that our MDE-based approach could handle in Section 3. Section 4 discusses our MDE-based approach for heterogeneous architectures. Preliminary results are in Section 5.

2 RELATED WORK

The significance of relying on DSLs to simplify the development of simulation models is clarified by many proposals that have appeared in the literature. One relevant example is BioNetGen, introduced by Blinov et al. (2004), which employs a rule-based modeling approach to address the combinatorial complexity arising from multiple molecular interactions and modifications in signal transduction systems. This approach provides an efficient way to manage the vast number of potential molecular states that can arise in biological signaling processes, simplifying the modeling of signal transduction at the molecular level. The model is written in a very concise but highly expressive way. The methodology used by BioNetGen has paved the way for the development of ML3, a language designed for agent-based discrete-event modeling and simulation of linked lives by Reinhardt et al. (2022). ML3 allows for the simulation of complex social interactions and life courses that are age-dependent and interconnected through social ties, providing a formal syntax and semantics based on Generalized Semi-Markov Processes. This approach enhances the accuracy and flexibility of simulations as well as supports efficient algorithmic implementations.

Similarly, OpenABL, a DSL tailored for agent-based modeling on parallel and distributed systems, has been introduced by Cosenza et al. (2018). OpenABL abstracts the complexity of parallel programming, allowing modelers to focus on the design and dynamics of simulations without digging into the low-level details of parallel execution. Xiao et al. (2020) extended OpenABL, proposing an enhanced automatic code generation framework for agent-based simulations on heterogeneous hardware platforms including CPUs, GPUs, and FPGAs.

A DSL explicitly targeting heterogeneous systems is Vivaldi by Choi et al. (2014). It is designed to streamline the process of volume processing and visualization on distributed heterogeneous systems. Vivaldi offers a Python-like syntax that simplifies the programming experience, making high-throughput parallel computing accessible to non-experts. It abstracts complex memory and execution management, which is automatically handled by its runtime system, thus freeing users from the intricacies of parallel programming. This enables efficient utilization of computing resources such as multi-core CPUs and GPUs without the need for detailed knowledge of underlying technologies like MPI or CUDA.

While all these DSLs are clearly useful for the end users, targeting heterogeneous architectures is not easy. Indeed, their implementations are handmade and would require non-minimal work to adapt to different (future) hardware. Moreover, if the target is the runtime orchestration of simulations, the required work for each language could be high and not necessarily re-usable. All these are aspects that we explicitly target with our MDE-based approach.

Code generation for heterogeneous platforms is targeted by Grewe et al. (2013), Li et al. (2015), Xiao et al. (2020), and Nguyen et al. (2019) by using methods such as pattern-matching to detect parallelizable C snippets, code templates, or high-level synthesis from OpenCL. These approaches generally focus on detecting localized sections of the source code that can be parallelized, or target specific domains. By relying on an MDE-based approach, we could extract more parallelism for multiple domains at the same time.

Another high-level model description is the Discrete Event System Specification (DEVS) (Zeigler et al. 2000). The beauty of DEVS is its mathematical foundation, which makes it useful for multiple purposes, from the description of the models to their verification. At the same time, being a mathematical formalism, it requires some different language or toolkit for proper implementation and execution of the model (Uhrmacher et al. 2024). One such example, the CD++ toolkit, has been proposed by Wainer (2002). The toolkit's support for modular model description facilitates reuse and system integration, reducing development time. Notably, Liu and Wainer (2010) have used CD++ to accelerate large-scale DEVS-based simulations on the heterogeneous Cell processor. This research line has shown the viability of improving simulation speeds by distributing workloads across multiple processing elements. At the same time, adapting the approach to other DSLs or porting the proposals to different (more complex) heterogeneous infrastructures may require non-negligible effort. The MDE-based approach we envision in this paper would also allow the CD++ toolkit (or other DEVS-based approaches) to be used as targets.

A relevant proposal in the literature is Delite (Sujeeth et al. 2014), a framework designed to facilitate the development of performance-oriented DSLs for heterogeneous systems. Delite's architecture simplifies the DSL development process by offering a reusable compiler infrastructure which supports the generation of optimized code for various hardware configurations, including CPUs and GPUs. It leverages Scala's advanced language features to enable embedded DSLs that utilize a sophisticated intermediate representation, allowing domain-specific optimizations and efficient execution across different hardware platforms. On top of it, Chafi et al. (2011) propose as an example OptiML, a DSL for machine learning that provides high-level abstractions for parallel computations. High-level application code is transformed into optimized parallel operations for heterogeneous devices without requiring source code modifications. The intermediate representation of Delite is a typical compiler-based intermediate representation. Conversely, we propose to rely on metamodels as an intermediate representation. This approach allows us to enlarge at will the number of supported transformations, thus enabling higher reuse also between the different DSL implementations.

The relevance and technical difficulty of transforming code to target heterogeneous computing is acknowledged by Baskaran et al. (2010), who have developed an automatic C-to-CUDA code generation tool for affine programs. Their research addresses the challenge of manually translating sequential C programs into parallel CUDA programs, which can be intricate and error-prone due to CUDA's complex memory hierarchy and multi-threaded model. This type of problem is exactly what we target using an MDE-based approach: metamodels could be leveraged to transform code between different incarnations, efficiently targeting differentiated hardware.

At the same time, having multiple versions of the same simulation model that could be run on different hardware platforms does not suffice the optimization goals of modern modeling and simulation. A fundamental aspect deals with orchestrating the execution of these different model versions when running a single simulation. O'Neal et al. (2022) discuss a domain-specific runtime designed to orchestrate computation on heterogeneous platforms, focusing on integrating domain-specific knowledge for scientific simulations. The proposed approach is aware of domain-specific operations and their data dependencies. The system facilitates more efficient utilization of heterogeneous computing resources, such as CPUs, GPUs, and other accelerators. This is achieved through a combination of offline tools for application configuration and an orchestration runtime subsystem that dynamically manages computation and data movement. Marotta et al. (2024) have taken this approach further, allowing the same simulation model to run on CPUs and GPUs for performance and energy efficiency reasons. The runtime orchestrator decides upon the best-suited hardware platform for a specific execution phase and synchronizes the state of the two families of devices transparently. Nevertheless, the models' code should be available for both CPUs and GPUs in these approaches. Generating efficient code for accelerators starting from a higher-level description is part of what we tackle in this paper.

3 SOURCES OF HETEROGENEITY AND BENEFITS OF MODEL-DRIVEN ENGINEERING

This section discusses the possible sources of heterogeneity that an MDE-based approach could consider.

Hardware heterogeneity can arise from various sources, including differences in processor architectures, memory systems, and interconnects within a single computing system. For instance, a system may have a mix of CPUs, GPUs, FPGAs, and other specialized processors, each with their own strengths and weaknesses. This is the simplest and most evident form of heterogeneity to deal with. From a technical perspective, developing efficient software that can run on different architectures and seamlessly switch from one another can be daunting.

Programming models also contribute to heterogeneity. Different programming models, such as MPI, OpenMP, and CUDA, may exploit different types of parallelism, such as message passing, shared memory, or data parallelism. These models may require different programming languages, syntax, and APIs, making it difficult to write portable, high-performance code that can run on multiple architectures.

Libraries and *support software* can also introduce heterogeneity. For example, different numerical libraries may be optimized for specific hardware architectures or programming models, so that choosing the best library for a given application is challenging. Similarly, simulation frameworks and tools may be designed for specific architectures or programming models, making it difficult to implement simulations that can run efficiently on a range of systems.

DSLs can help hide away these sources of heterogeneity from model developers. Nevertheless, from the point of view of the DSL designer, the situation is not easy. The designer must ensure that the DSL is optimized for each architecture, which may require significant knowledge of hardware and software optimization techniques. If multiple DSLs have to be developed, the same repeating (but highly technical) challenges will be faced.

Conversely, an MDE approach can simplify the development of (multiple) DSLs for heterogeneous architectures by providing a systematic way of creating high-level abstractions that capture the essence of the architecture and its components. One of the main benefits of the MDE approach is that it allows for the reuse of developed assets. Once a model has been created and validated, it can generate code for multiple DSLs. This reduces the time and effort required to develop new DSLs and ensures consistency across different DSLs. Additionally, the MDE approach allows for the automatic generation of code optimized for the heterogeneous architecture being used, which can lead to significant performance gains.

Technical aspects of the MDE approach include the use of modeling languages such as UML, SysML, or DSLs themselves to create models of the architecture and its components. These models can then be transformed into code using model transformation languages. The generated code can be further optimized using more traditional compiler optimizations.

4 METAMODELING FOR PDES ON HETEROGENEOUS ARCHITECTURES

We show in Figure 1 the general architecture that we envisage to support the generation of simulation model executables that can be run on top of heterogeneous architectures—we refer to these executables as *hybrid binaries*. The simulation expert writes a model in a certain DSL. A model-to-model transformation generates an intermediate representation (IR) of the model. Then, text-to-text transformations generate the variants of equivalent source code of the original model, targeting diverse hardware architectures. At this point, standard compilers for the architectures can be used to generate binary representations of the model. These can be linked together to generate the hybrid binary.

Additional aspects are relevant for generating a hybrid binary that can orchestrate the execution of the model on heterogeneous architectures, moving parts of the model's execution on the available hardware. First, this runtime orchestration requires knowing how to move the data across the various available hardware instances. For this purpose, the shape and organization of the simulation model state must be known by the runtime environment. Because we can work at the metamodel level, some additional transformations can be used for this purpose. The *data dependency tracker* is a component of our architecture that performs additional model-to-text transformations to output what we call an *access trace* file. This access trace describes at what points of execution the simulation model allocates memory buffers or where variable updates are performed. This information can be represented in any way that is suitable for the runtime
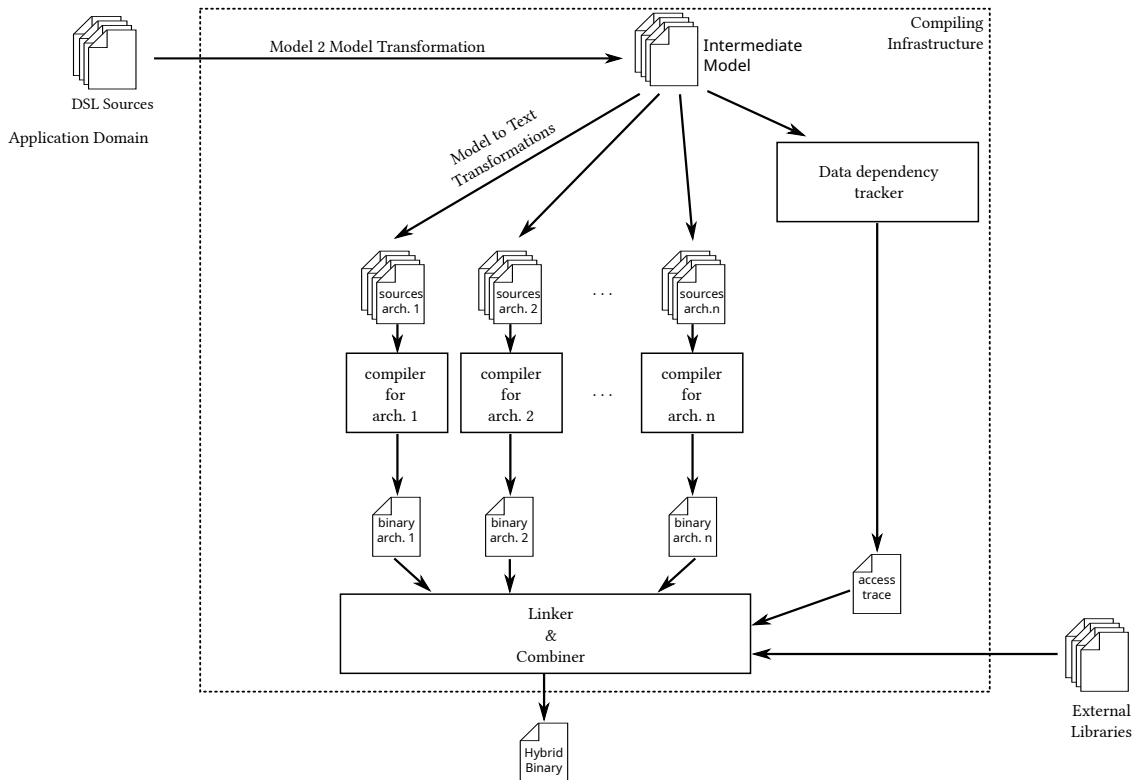
Figure 1: Generation of Simulation Models from DSLs using an MDE Approach.

orchestrator to track the updates to the simulation state. Notable examples are already present in the literature for simulation runtime environments that track the simulation state evolution over time for housekeeping operations (see, e.g., Steinman 1993; West and Panesar 1996; Pellegrini et al. 2015).

Additional components should be linked together to produce a fully working hybrid binary. First and foremost, the runtime/orchestrator. It could be a monolithic runtime environment, or it may be composed of, e.g. different runtimes for CPUs, GPUs, and FPGAs, all glued together by some orchestration logic—this is, for example, the approach taken by Marotta et al. (2024). From the perspective of the metamodeling approach, it does not change much. The metamodel for the IR should at least allow to describe the following:

- A *portion* of the simulation model. The simulation object/logical process abstractions (Fujimoto 1990a), for example, are perfectly suitable here. This portion of the model is a process component that can be logically moved around the available hardware at runtime. It is logically associated with all the events the object is in charge of processing.
- *Simulation runtime interface*. It is the API exposed by the simulation runtime/orchestrator that the model can implicitly or explicitly invoke. Also, the API related to *explicit migration* of the objects or to trigger some *object allocation* across the different hardware instances can be considered.
- *Library interfaces*. They allow the representation of specific support exploited by the model and implemented in third-party libraries, or they can relate to specific capabilities of the runtime system. For example, we could use our metamodeling approach to generate reverse events (Carothers et al. 1999) on the fly, using approaches similar to what LaPre et al. (2014) did at the compiler IR level.

We show in Figure 2 the UML representation of the main components of our IR metamodel. The *DES Model* is the root concept of our IR metamodel, which is essentially a composition of the minimal elements required to describe a Discrete Event Simulation model: *struct definitions*, *variables* and *event handlers*.

Struct definitions were introduced to simplify the representation of concepts like custom types, event payloads, and simulation object states—Figure 2 is simplified and does not adequately represent all required
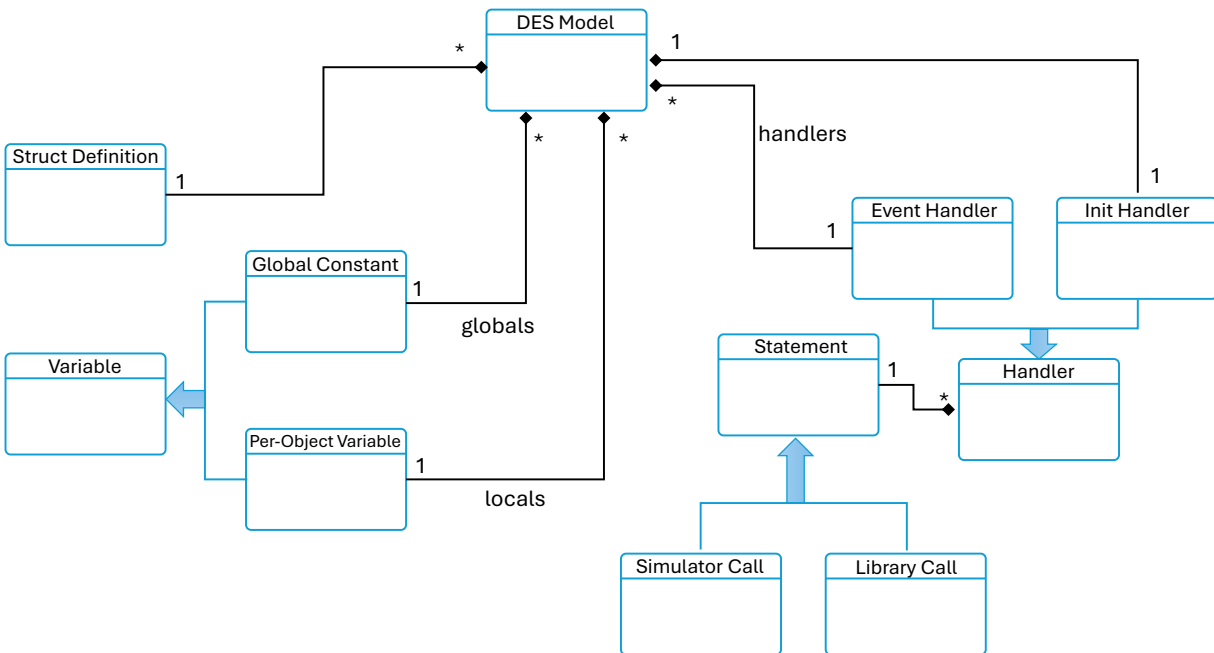
Figure 2: Simplified UML class diagram of the proposed Intermediate Representation metamodel.

concepts, such as fields and field types. The variables in a DES Model can be of two types: *global* and *per-object*. The former represent simulation model parameters that are usually accessible from any simulation object and, for this reason, are usually read-only—as future development, we plan to include techniques for reversible global variables management, similarly to the proposal of Pellegrini et al. (2016), at metamodel level. Conversely, per-object variables are read/write, representing the state of each simulation object.

Handlers are procedures, namely sequences of statements, responsible for updating/accessing simulation variables. We distinguish between initialization and regular event handlers. Statements are strings belonging to a functional language used to express operational semantics, which, in our case, is a simplified version of C. In our IR language, two types of function calls play a crucial role: *simulator* and *library* calls. Simulator calls describe the interface of an abstract DES simulator (e.g., creating and sending a new event). Conversely, library calls represent any third-party software module (e.g., pseudo-random number generators) the simulation model requires. Having both explicitly represented in our IR language allows us to easily cope with them when targeting different simulation platforms and environments.

Transforming the IR model into a text is a two-step process. In the first step, the IR model is transformed into a standard C model, where both simulator and library calls are resolved to C calls targeting specific software libraries using user-provided bindings. This approach allows us to simplify translating DSL-defined models to different simulation platforms. For instance, when a simulator-call binding is defined for a specific DES simulator, the same binding can be reused to translate any IR model to a source targeting that platform. Secondly, the C model is transformed into a text, which is the final code to compile.

5 EXPERIMENTAL RESULTS

We evaluate here the proof of concept of the proposed approach: Section 5.1 describes the PDES engines and the orchestrator we used; Section 5.2 presents the model written in our IR language that we used as an application example; we discuss the results of our experimental evaluation in Section 5.3.

5.1 Experimental Setup

To exercise our MDE approach, we relied on the recent "follow the leader" (FTL) orchestrator (Marotta et al. 2024). It is an interoperability and optimization layer between two different speculative PDES engines based on the Time Warp synchronization protocol by Jefferson (1985), namely ROOT-Sim (CPU-based) by Pellegrini et al. (2012) and GPUTW (targeting GPUs) by Liu and Andelfinger (2017). When provided with a C and CUDA version of the same simulation model, FTL periodically synchronizes the simulation state and starts a "challenge" between simulations run on CPUs and GPUs. After a short execution on both, a performance estimation for the near future is carried out, and the best-suited hardware architecture continues to run the simulation.

We used the classical PHold model (Fujimoto 1990b) generated with the MDE approach and used it with two execution phases: *balanced* and *unbalanced*. FTL's orchestration capabilities benefit from cycling through these phases as CPU/GPU simulators respond differently to them. We ran on a heterogeneous CPU/GPU node equipped with an Intel i7-12700H, an Nvidia RTX 3060, and 64GB of RAM.

5.2 PHold Model

```
 1 phold {
 2   description: "this is an alternating balanced/unbalanced phold"
 3   struct definitions:
 4     state_t {
 5       int me
 6       int rcv_evts
 7       rand_t seed
 8     }
 9
10   global constants:
11     double lambda = 1.0
12     double hot_fraction = 0.01
13     double phase_window_size = 8000000
14     double end_sim_gvt = 60000000
15
16   model state variable:
17     state_t lp_state
18
19   handler INIT:
20     lp_state.rcv_evts = lp_state.rcv_evts + 1
21     SendEvent EVT to MySelf() at now+rnd::Exponential(seed, lambda) with NULL
22
23   handler EVT:
24     lp_state.rcv_evts = lp_state.rcv_evts + 1
25     int dest = GetRandomObject()
26     int cur_hot_phase = (now / phase_window_size)
27     int unbalanced = cur_hot_phase%2;
28     if(unbalanced)
29       dest = dest % ( hot_fraction * GetNumObjects())
30     SendEvent EVT to dest at now+rnd::Exponential(seed, lambda) with NULL
31     return now >= end_sim_gvt
32
33 }
```

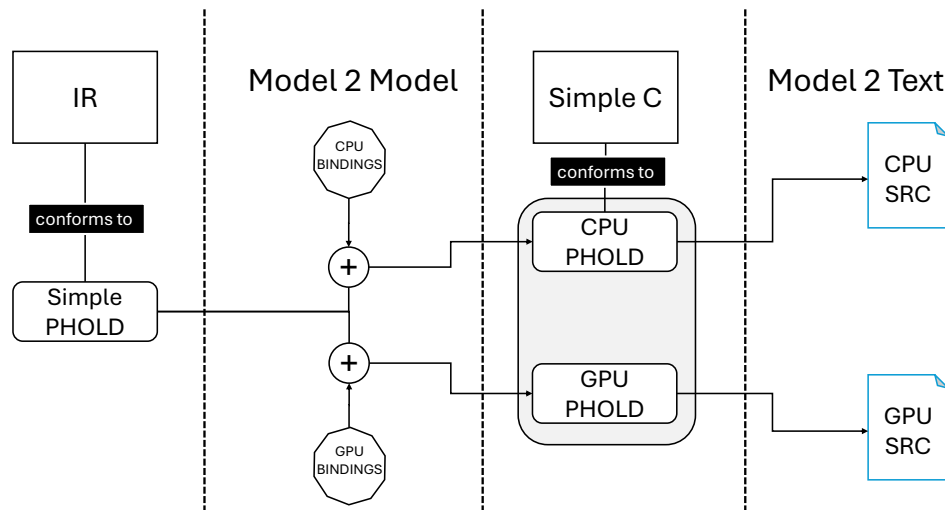Figure 3: Intermediate Representation Model of Alternating PHold.

Figure 4: Scheme of transformations used to generate CPU/GPU source code from IR PHold model.

Figure 3 shows the PHold model we used as a use-case. It is written directly in our IR language and implemented within the IntelliJ MPS modeling framework (Pech 2021). Lines 3-7 define a new struct used as a simulation object state (see lines 15–16), essentially a counter of the received events and an integer representing the simulation object identifier. Lines 9–13 define the global constants accessible by handlers. Finally, lines 18-30 provide the actual code of handlers. The `INIT` handler simply increments the counter and sends an event of type `EVT` to the simulation object currently running the `INIT` handler. This makes the simulation start by injecting the very first events at line 20. Line 21 simulator and library calls, namely `SendEvent`, `Myself`, `now` and `rnd::Exponential`. The `EVT` handler describes the behavior of a simulation object when it receives an `EVT` event. In particular, it increments the counter of received events and defines the simulation object that should be the target of the next `SendEvent`. To do so, it first detects the current simulation phase. If *balanced*, any simulation object can receive the new event. Conversely, if *unbalanced*, the newly generated event can be sent to a small subset of the simulation object, whose size is the 1% of the simulation objects.

The process to transform the PHold model written in our IR language to source code compatible with both the target PDES engines is summarized in Figure 4. As described in Section 4, it runs in two phases. In the first, we exploit a model-to-model transformation to generate a pair of simulation models conforming to a metamodel, which describes standard C/CUDA programs. Each model targets a different PDES engine: a CPU engine (ROOT-Sim) and a GPU engine (GPU-TW). The transformation takes an additional model representing bindings between simulator/library calls in the IR model and actual C/CUDA function calls. It also maps IR types to simulator/library types. This step is crucial to resolve the idiosyncrasy of different simulators and platforms. For instance, `SendEvent` calls are mapped to calls to schedule a new event for each simulator, and `Exponential` calls are mapped to the proper CPU/GPU `CuRand` implementation. Since the target metamodel represents C source files, we exploit this stage to add proper includes to reference the required libraries—the definition of those bindings is reusable across different IR models.

Once the bindings are complete, we proceed with the last stage, which transforms both CPU/GPU models into text with proper formatting, namely C files that can be compiled against each PDES platform.

5.3 Results

We have replicated the same experiments in (Marotta et al. 2024). The results, perfectly compatible with the original publication, are shown in Figure 5. In the balanced configuration, the GPU version significantly

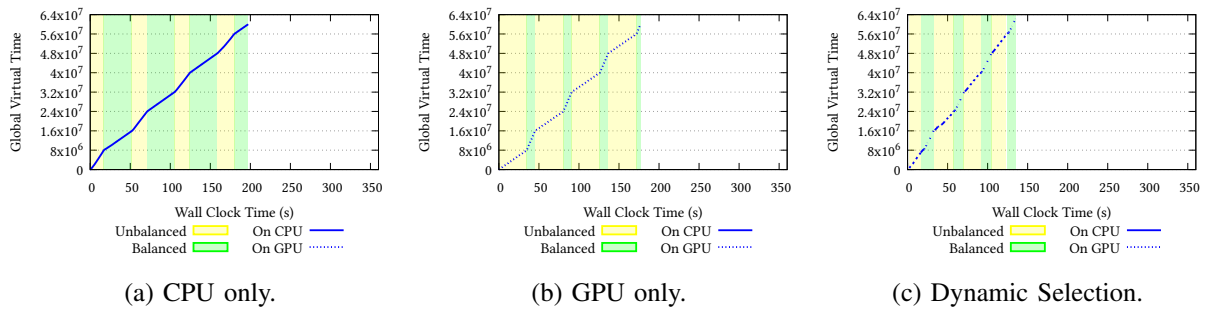(a) CPU only. (b) GPU only. (c) Dynamic Selection.

Figure 5: Performance Results using FTL. The orchestrator runs every 2 wall-clock seconds.

outperforms the CPU version. Conversely, the unbalanced configuration favors the CPU implementation. The FTL orchestrator can effectively capture these dynamics, thus showing a higher overall performance, by promptly switching across the two implementations. For space constraints, we cannot discuss energy-related results, but we essentially observe no significant discontinuity with the energy profile of CPU-only or GPU-only simulations, providing a performance improvement for free. We refer the reader to the original paper for a more thorough discussion on this aspect.

These results are relevant because they confirm the ability of our MDE-based approach to generate source code variants that can be compiled to a hybrid binary plugging existing runtime simulation environments, able to orchestrate the execution of a single simulation model on heterogeneous architectures. Starting from a simple DSL, the burden of generating optimized code for GPUs and CPUs is demanded for multiple model-to-model and model-to-text transformations.

6 CONCLUSIONS AND FUTURE WORK

This work has explored how MDE can optimize parallel discrete event simulations on heterogeneous architectures. By abstracting hardware complexities, MDE enables simulation experts to focus on model fidelity, optimizing the use of varied hardware and enhancing the efficiency and scalability of simulations. Domain-specific languages integrated with MDE provide a framework that reduces development time and enhances performance, setting the groundwork for dynamic, adaptive simulation frameworks that can harness the evolving capabilities of heterogeneous computing environments.

In future work, we plan to target additional sources of heterogeneity identified in Section 3, also focusing on the interoperability of simulation models with different runtime environments or orchestrators. We also plan to broaden the set of hardware accelerators supported by our MDE approach (e.g., FPGAs), to benefit from more diverse execution capabilities, both in terms of performance and energy efficiency.

ACKNOWLEDGMENTS

REFERENCES

Baskaran, M. M., J. Ramanujam, and P. Sadayappan. 2010. "Automatic C-to-CUDA code generation for affine programs". In *Compiler Construction*, edited by R. Gupta, Volume 6011 LNCS of *Lecture notes in computer science*, 244–263. Berlin, Heidelberg: Springer https://doi.org/10.1007/978-3-642-11970-5_14.

Blinov, M. L., J. R. Faeder, B. Goldstein, and W. S. Hlavacek. 2004. "BioNetGen: software for rule-based modeling of signal transduction based on the interactions of molecular domains". *Bioinformatics (Oxford, England)* 20(17):3289–3291 https://doi.org/10.1093/bioinformatics/bth378.

Blunk, A. and J. Fischer. 2013. "Efficient development of domain-specific simulation modelling languages and tools". In *Lecture Notes in Computer Science*, Lecture notes in computer science, 163–181. Berlin, Heidelberg: Springer Berlin Heidelberg https://doi.org/10.1007/978-3-642-38911-5_10.

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. 1999. "Efficient Optimistic Parallel Simulations Using Reverse Computation". *ACM Transactions on Modeling and Computer Simulation* 9(3):224–253 https://doi.org/10.1145/347823.347828.

Chafi, H., A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya and K. Olukotun. 2011. "A domain-specific approach to heterogeneous parallelism". *ACM SIGPLAN Notices* 46(8):35–46 https://doi.org/10.1145/2038037.1941561.

Choi, H., W. Choi, T. M. Quan, D. G. C. Hildebrand, H. Pfister and W.-K. Jeong. 2014. "Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems". *IEEE transactions on visualization and computer graphics* 20(12):2407–2416 https://doi.org/10.1109/TVCG.2014.2346322.

Cosenza, B., N. Popov, B. Juurlink, P. Richmond, M. K. Chimeh, C. Spagnuolo, *et al*. 2018. "OpenABL: A domain-specific language for parallel and distributed agent-based simulations". In *Euro-Par 2018: Parallel Processing*, edited by M. Aldinucci, L. Padovani, and M. Torquati, Lecture notes in computer science, 505–518. Cham: Springer International Publishing https://doi.org/10.1007/978-3-319-96983-1_36.

Fujimoto, R. M. 1990a. "Parallel Discrete Event Simulation". *Communications of the ACM* 33(10):30–53 https://doi.org/10.1145/84537.84545.

Fujimoto, R. M. 1990b. "Performance of Time Warp Under Synthetic Workloads". In *Distributed Simulation*, edited by D. Nicol, PADS'90, 23–28. San Diego, CA, USA: Society for Computer Simulation International.

Gagliardi, F., M. Moreto, M. Olivieri, and M. Valero. 2019. "The international race towards Exascale in Europe". *CCF transactions on high performance computing* 1(1):3–13 https://doi.org/10.1007/s42514-019-00002-y.

Grewe, D., Z. Wang, and M. F. P. O'Boyle. 2013. "Portable mapping of data parallel programs to OpenCL for heterogeneous systems". In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, GCO'13, 1–10. Piscataway, NJ, USA: IEEE https://doi.org/10.1109/cgo.2013.6494993.

Jameson, A. and J. C. Vassberg. 2001. "Computational fluid dynamics for aerodynamic design: Its current and future impact". In *Proceedings of the 39th AIAA Aerospace Sciences Meeting & Exhibit*, 1–26. Reston, VA, USA: American Institute of Aeronautics & Astronautics https://doi.org/10.2514/6.2001-538.

Jefferson, D. R. 1985. "Virtual Time". *ACM Transactions on Programming Languages and Systems* 7(3):404–425 https://doi.org/10.1145/3916.3988.

Kothe, D., S. Lee, and I. Qualters. 2019. "Exascale computing in the United States". *Computing in science & engineering* 21(1):17–29 https://doi.org/10.1109/mcse.2018.2875366.

LaPre, J. M., E. J. Gonsiorowski, and C. D. Carothers. 2014. "LORAIN: a step closer to the PDES 'holy grail'". In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS'14, 3–14. New York, NY, USA: ACM https://doi.org/10.1145/2601381.2601397.

Li, P., E. Brunet, F. Trahay, C. Parrot, G. Thomas and R. Namyst. 2015. "Automatic OpenCL code generation for multi-device heterogeneous architectures". In *2015 44th International Conference on Parallel Processing*: IEEE https://doi.org/10.1109/icpp.2015.105.

Liu, Q. and G. Wainer. 2010. "Accelerating large-scale DEVS-based simulation on the cell processor". In *Proceedings of the 2010 Spring Simulation Multiconference*. San Diego, CA, USA: Society for Computer Simulation International https://doi.org/10.1145/1878537.1878667.

Liu, X. and P. Andelfinger. 2017. "Time Warp on the GPU: Design and Assessment". In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '17, 109–120. New York, NY, USA: ACM https://doi.org/10.1145/3064911.3064912.

Marotta, R., A. Pellegrini, and P. Andelfinger. 2024. "Follow the Leader: Alternating CPU/GPU Computations in PDES". In *Proceedings of the 2024 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '24. Atlanta, GA, USA: ACM https://doi.org/10.1145/3615979.3656056.

Montesano, F., R. Marotta, and F. Quaglia. 2022. "Spatial/Temporal Locality-based Load-sharing in Speculative Discrete Event Simulation on Multi-core Machines". In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '22, 81–92. New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3518997.3531026.

Nguyen, Q. A. P., P. Andelfinger, W. Cai, and A. Knoll. 2019. "Transitioning Spiking Neural Network Simulators to Heterogeneous Hardware". In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS, 115–126. New York, NY, USA: ACM https://doi.org/10.1145/3316480.3322893.

O'Neal, J., M. Wahib, A. Dubey, K. Weide, T. Klosterman and J. Rudi. 2022. "Domain-specific runtime to orchestrate computation on heterogeneous platforms". In *Euro-Par 2021: Parallel Processing Workshops*, Lecture notes in computer science, 154–165. Cham: Springer International Publishing https://doi.org/10.1007/978-3-031-06156-1_13.

Pech, V. 2021. "JetBrains MPS: Why modern language workbenches matter". In *Domain-Specific Languages in Practice*, 1–22. Cham: Springer International Publishing https://doi.org/10.1007/978-3-030-73758-0_1.

Pellegrini, A., S. Peluso, F. Quaglia, and R. Vitali. 2016. "Transparent speculative parallelization of discrete event simulation applications using global variables". *International journal of parallel programming* 44(6):1200–1247 https://doi.org/10.1007/s10766-016-0429-2.

Pellegrini, A., R. Vitali, and F. Quaglia. 2012. "The ROme OpTimistic Simulator: Core Internals and Programming Model". In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTOOLS, 96–98. Brussels, Belgium: ICST https://doi.org/10.4108/icst.simutools.2011.245551.

Pellegrini, A., R. Vitali, and F. Quaglia. 2015. "Autonomic State Management for Optimistic Simulation Platforms". *IEEE Transactions on Parallel and Distributed Systems* 26:1560–1569 https://doi.org/10.1109/TPDS.2014.2323967.

Reinhardt, O., T. Warnke, and A. M. Uhrmacher. 2022. "A language for agent-based discrete-event modeling and simulation of Linked Lives". *ACM transactions on modeling and computer simulation: a publication of the Association for Computing Machinery* 32(1):1–26 https://doi.org/10.1145/3486634.

Rodrigues da Silva, A. 2015. "Model-driven engineering: A survey supported by the unified conceptual model". *Computer languages, systems & structures* 43:139–155 https://doi.org/10.1016/j.cl.2015.06.001.

Steinman, J. S. 1993. "Incremental State Saving in SPEEDES Using C Plus Plus". 687–696: Society for Computer Simulation.

Sujeeth, A. K., K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky *et al*. 2014. "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages". *ACM Transactions on Embedded Computing Systems* 13(4s):1–25 https://doi.org/10.1145/2584665.

Sutter, H. 2014. "Welcome to the Jungle: Or, a Heterogeneous Supercomputer in Every Pocket". Technical report, Sutter's Mill: Herb Sutter on software development.

Taffoni, G., L. Tornatore, D. Goz, A. Ragagnin, S. Bertocco, I. Coretti, *et al*. 2019. "Towards exascale: Measuring the energy footprint of astrophysics HPC simulations". In *2019 15th International Conference on eScience (eScience)*: IEEE https://doi.org/10.1109/escience.2019.00052.

Uhrmacher, A., P. Frazier, R. Hähnle, *et al*. 2024. "Context, composition, automation, and communication - the C 2 AC roadmap for modeling and simulation". *ACM Transactions on Modeling and Computer Simulation*.

Wainer, G. 2002. "CD++: A toolkit to develop DEVS models". *Software—Practice and Experience* 32:1261–1306 https://doi.org/10.1002/spe.482.

Warnke, T. 2020. "Domain-specific languages for modeling and simulation". Master's thesis, Universität Rostock, Rostock, Germany https://doi.org/10.18453/ROSDOK_ID00002966.

West, D. and K. Panesar. 1996. "Automatic Incremental State Saving". In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, PADS, 78–85. Piscataway, NJ, USA: IEEE https://doi.org/10.1109/PADS.1996.761565.

Xiao, J., P. Andelfinger, W. Cai, P. Richmond, A. Knoll and D. Eckhoff. 2020. "OpenABLext: An automatic code generation framework for agent-based simulations on CPU-GPU-FPGA heterogeneous platforms". *Concurrency and computation: practice & experience* 32(21):1–15 https://doi.org/10.1002/cpe.5807.

Xiao, J., G. Kilinç, P. Andelfinger, D. Eckhoff, W. Cai and A. Knoll. 2020. "Pedal to the Bare Metal: Road Traffic Simulation on FPGAs Using High-Level Synthesis". 117–121: ACM https://doi.org/10.1145/3384441.3395979.

Zahran, M. 2017. "Heterogeneous computing: here to stay". *Communications of the ACM* 60(3):42–45 https://doi.org/10.1145/3024918.

Zeigler, B. P., T. G. Kim, and H. Praehofer. 2000. *Theory of Modeling and Simulation*. London, UK: Academic Press https://doi.org/10.1016/C2016-0-03987-6.

AUTHOR BIOGRAPHIES

ROMOLO MAROTTA received a Ph.D. in Computer Engineering from Sapienza University of Rome. He is a researcher at the University of Rome Tor Vergata and has been a postdoctoral researcher at the University of L'Aquila. His research activities mainly focus on concurrent data structures, synchronization algorithms, operating systems and parallel simulation. His email address is r.marotta@ing.uniroma2.it and his website is https://romolomarotta.github.io/.

ALESSANDRO PELLEGRINI received a PhD degree in Computer Engineering from Sapienza, University of Rome in 2014. His research interests are in the simulation of parallel and distributed architectures, a field in which he has more than 100 publications in books, journals and conference proceedings. His email address is a.pellegrini@ing.uniroma2.it and his website is https://alessandropellegrini.it.