# Automating Change Propagation in Software Applications for Heterogeneous Computing

Simone Bauco
Tor Vergata University of Rome
Rome, Italy
simone.bauco@uniroma2.it

Guglielmo De Angelis
IASI-CNR
Rome, Italy
guglielmo.deangelis@iasi.cnr.it

Alessandro Pellegrini
Tor Vergata University of Rome
Rome, Italy
a.pellegrini@ing.uniroma2.it

***Abstract*—Developing applications for heterogeneous systems is challenging because maintaining semantic consistency between CPU and GPU code bases is difficult. We address this by proposing a software architecture grounded on Model-Driven Engineering that isolates logical models from physical implementations. Our approach transforms a high-level Domain-Specific Language into an intermediate representation based on the Actor Model, enabling the automatic generation of both C and CUDA code from a single source. We evaluate the framework using Parallel Discrete Event Simulation benchmarks. Results confirm that the architecture successfully propagates functional changes to all targets, ensuring the synthesised code is highly performant and functionally equivalent across platforms.**

***Index Terms*—Change Propagation, MDE, Software Architecture, Actor Model, Heterogeneous Architectures**

## I. Introduction

Heterogeneous computing [1, 2] is a modern computing paradigm that has enabled the development of exascale supercomputers [3, 4]. High-performance computations are achieved by leveraging diverse hardware devices installed on the same machine, allowing one to benefit from the increased processing power naturally provided by the different hardware architectures of the underlying machines.

Typical devices used in heterogeneous computing include Non-uniform memory access (NUMA), CPUs, Graphics processing units (GPUs), and various types of accelerators, such as Field-programmable gate arrays (FPGAs), Tensor Processing Units (TPUs), and similar. From a software development standpoint, the major hindrance to execution on heterogeneous computers stems from the different programming models they offer. Indeed, while CPUs support most programming paradigms, when designing code to run on GPUs or accelerators, specific activities must typically be isolated in ad-hoc code blocks that are offloaded to the target device (e.g., CUDA kernels for GPUs or VHDL code for FPGAs). This requires organising the application for explicit exploitation of the available hardware, making the development of applications for heterogeneous computers more difficult than traditional applications [5, 6, 7]. At the same time, failing to provide this organisation of the software may render exascale systems useless, as they are inherently heterogeneous.

A notable class of software for heterogeneous computers is task-based applications [8]. Here, the logical unit of the application is the task, which is a sequence of activities typically executed reactively, i.e., when some conditions occur, namely when a task is scheduled. Task-based applications are effectively designed using an event-driven approach [9]: whenever an event is encountered, an associated handler is activated to handle it.

This class of applications is interesting from a heterogeneous computing perspective because it already captures the application's global behaviour in event handlers. These handlers are typically activated by a runtime environment, thus decoupling the application design from the application runtime. In this sense, a single logical event handler can have multiple implementations, targeting the different devices (and Instruction Set Architectures – ISAs) that compose a heterogeneous computer—something that we call a *heterogeneous executable*. From an execution point of view, the focus can shift: rather than explicitly coding the activation of, e.g., kernels on GPUs, it becomes an *orchestration problem* for the runtime environment. In other words, by having multiple implementations of the same event handler logic, the underlying runtime environment must decide which device (or devices) some (batch of) events should be executed on.

From a software engineering perspective, the precondition for runtime orchestration is already hard. Complex compilation toolchains can generate heterogeneous executables (e.g., [10]) that emit machine code for different ISAs and glue together different versions into a single executable. For that to work, different *equivalent* versions of the same handler logic must be written in different programming languages (e.g., C, CUDA, Verilog). Proving the equivalence of all these versions is undecidable in general, or very hard for specific cases; but in the context of traditional software life cycles, the complexity is even greater: every change to the logic associated with the handler must be manually propagated across the different implementations. Maintaining the coherence between the functional specification and the different implementations targeting heterogeneous devices can be a daunting task. Anyhow, this task is strictly necessary: if the versions are not kept in alignment, any orchestration activity becomes pointless, as activating a single event handler on different devices may yield different outputs, thus rendering the overall execution incorrect.

Our work proposes a reference software architecture for heterogeneous computing environments [11] that leverages the core principles of Model-Driven Engineering (MDE) [12]. These principles suggest explicitly adopting several layers that separate the *logical aspects* of an application from its *physical implementation*. Logical layers aim to capture the core behaviour and rules of the application domain, independent of the hardware on which it will run, while the physical layers address the specific requirements of the underlying platform, such as distinct memory management and execution models found in CPUs and GPUs. By modelling these aspects individually, we can define the application structure once and rely on the referenced software architectural style to bridge the gap between high-level logic and low-level hardware details. This separation allows developers to focus on the problem domain rather than the intricacies of heterogeneous computing.

In this paper, we present a software architecture that enables the synthesis of executable code for applications targeting heterogeneous computing environments. The automatic generation process guarantees functional equivalence and automates change propagation across heterogeneous targets. In detail, our approach uses a transformation pipeline to move from abstract design to specific software artefacts. First, a Model-to-Model (M2M) transformation converts the logical specification of the simulation model into an intermediate representation based on the Actor Model [13, 14]. This step simplifies the system by breaking it down into independent interacting units. Subsequently, Model-to-Text (M2T) transformations automatically generate the final source code for specific hardware platforms. In particular, we use the M2T transformation initially presented in [11] to generate C code targeting CPUs, and we also propose a new M2T transformation in this paper that generates CUDA code targeting GPUs. Both artefacts are generated from the exact same source model, thanks to the design concepts defined by the referred architectural style (i.e., the Actor Model intermediate representation).

Finally, we show how this pipeline maintains software consistency when changes are applied. Typically, a simulation model evolves significantly over time [15], and manually maintaining separate versions for CPUs and GPUs could lead to subtle inconsistencies or misalignment between them. Using the proposed approach, a change to the model's logical specification is automatically propagated to the concrete artefacts for both architectures, ensuring synchronisation while removing the burden of manual updates and the risk of introducing errors. We also show that using two different concrete artefacts allows us to run simulations on CPUs and GPUs in a single run, effectively harnessing the compute power of modern heterogeneous architectures.

To evaluate our proposed software architecture, we rely on a Domain-Specific Language (DSL), named DESL, tailored for discrete-event simulation models [16]. Specifically, we refer to DESL implementations of two different simulation models from the literature, namely PHold [17] and COMPADS [18].

The remainder of this paper is structured as follows. In Section II, we describe a proposal of a software architecture for heterogeneous computing environments initially presented in [11], on which we base our current work. Section III introduces the M2T transformation used to generate the GPU artefacts. In Section IV, we illustrate the details of our case study and report on the experimental assessment. Related work is discussed in Section V. Section VI draws the conclusions and illustrates future work.

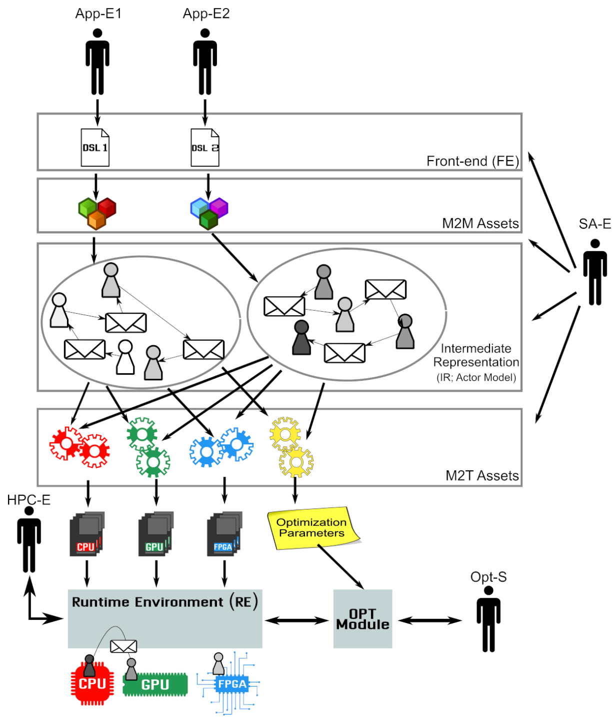

Fig. 1: Overall organisation of the Domain platform.

## II. Overall Software Architecture

This section briefly describes the Domain software platform [11]. Specifically, we describe the main architectural pillar offered by Domain, while we refer the reader to [11] for a comprehensive presentation of its potential stakeholders, and their foreseen roles.

Domain's architecture (which it is sketched in Fig. 1) is organised into several interconnected logical components, each one targeting a specific concern: the development of a specific task-based application (i.e., the Front-End – FE), the orchestration of the underlying hardware resources (i.e., *runtime environment* – RE), the definition of optimal policies to be followed while executing the software application of the underlying hardware infrastructure (i.e., the *optimiser* – OPT).

FE supports the adoption of domain-specific languages, easing the development of task-based applications. FE leverages MDE principles to enable artefact reusability across different hardware resources. In this sense, internally Domain relies on an intermediate representation (IR) grounded on the Actor Model [19]. In this architectural style, concurrent applications are modelled as *actors* that exchange messages. Each actor is responsive to receiving a new message and can generate new ones destined for any actor declared in the model (including itself). Thus, such IR is referred to as the target language for M2M transformations from any DSL referred to in the FE. In other words, any task-based application modelled with its specific DSL is converted into an IR artefact encoding the (dynamic) topology of actors interacting via message passing.

Certainly, adopting a new DSL requires defining a new M2M transformation from an application-oriented model to the Domain IR. However, such new task-based applications can benefit from assets (e.g., M2T transformations) developed in the context of other DSLs that can execute or optimise IR artefacts on heterogeneous architectures. For example, in Sec IV we will reuse the same M2T transformation (to generate C code targeting CPUs) that has been previously proposed in [11] for a different case study (with its own DSL). Conversely, in the absence of IR, the development phase of a new DSL would require not only the realisation of the language itself, but also the development of specific M2Ts to generate the low-level code. Considering the heterogeneous nature of the hardware infrastructure we tackle, this could require implementing multiple M2T transformations, one for each target hardware platform.

RE is responsible for orchestrating the correct execution of low-level code on differentiated (distributed) hardware instances. To meet this objective, the RE relies on inter-device task queues (implemented as custom $k$-heaps) to exchange messages generated by actors even when they are (currently) running on different hardware devices. These queues can be conveniently designed by HPC experts (HPC-Es) to capture the specific capabilities of the underlying hardware and the characteristics of the applications, thanks to their specification in the IR. Thus, each application developed using a DSL in the FE is first processed into hardware-specific bundles by means of a chain of transformations (e.g., M2M from the DSL to the IR, and then M2T from IR to the target platform) so that they can be deployed on top of the available hardware resources.

According to Fig. 1, the orchestrations in the RE are also affected by input from the OPT. Such a component is responsible for determining an optimal strategy to achieve some non-functional goal, such as maximising application performance, improving energy efficiency, or achieving a certain level of performance under a power cap [20]. As detailed in [21], the current reference implementation for OPT is based on *Answer Set Programming* [22] rules to specify the logical constraints that the allocation must satisfy. Although the detailed presentation of OPT or its reference implementation is not central for this work, we report that, given set of observed information on both the current status of the computation (i.e., application, and underlying hardware platform), an ASP solver computes an optimal placement of actors on the different devices composing the heterogeneous system, which is then suggested to RE.

## III. The CUDA M2T Transformation

In [11], we introduced the *ActorLanguage*, a reference implementation for IR that has been developed using JetBrains MPS [23]. An *ActorScript* is an instance of a task-based application implemented in ActorLanguage. In line with the workflow in Figure 3, a given ActorScript can be transformed into a collection of target artefacts encoded in lower-level models, such as the programming language for a specific hardware platform. Using any of these transformations, the core concepts of ActorLanguage are first translated into programs for a specific RE, which usually leverage hardware-specific libraries; these programs are then compiled into executable binaries that can be run on the specific architectures.

Among the others, we focused on developing M2T assets targeting ROOT-Sim [24], a parallel/speculative RE that orchestrates state-disjoint jobs using an event-driven programming approach. Specifically, in [11] we presented an M2T asset that transforms ActorScripts into C code for compilation and execution by ROOT-Sim on traditional CPUs. In the rest of this Section, we present the M2T that generates CUDA code that ROOT-Sim executes on GPUs.

In ROOT-Sim, a model is a set of callback functions that process events (i.e., event handlers) and initialisation functions that set up the simulation state. Our M2T transformation therefore generates a `main` function that configures the simulation model's execution (on both CPUs and GPUs) by simply including the RE's main header file and adhering to ROOT-Sim's configuration struct. In the configuration, the set of logical processes that the RE must allocate and orchestrate is also defined, mapping each actor declared in the ActorScript to an LP. Also, because ROOT-Sim requires that each logical process have a unique identifier, the address of each actor is used to generate the ID of its corresponding logical process.

Then, the M2T generates specific structs for the simulation state and the event payloads used throughout the simulation. Each LP class may have a different simulation state, so a specific state struct is emitted by iterating over each actor's ActorScript definition. Similarly, the M2T synthesises the declaration of a different struct that maintains the message instances exchanged by the actors. Notably, both these structs trace information that has been originally specified in terms of some specific DSL (i.e., in the FE of Figure 1) and then encoded in the IR based on the Actor Model by means of M2M transformations (see Figure 1).

Unlike CPUs, the GPU version requires memory to be pre-allocated for kernel access. The generation process for CUDA targets avoids using dynamic pointers, which would introduce significant overhead and memory fragmentation on the device. Instead, the M2T flattens the Actor Model's state and message structures into a contiguous memory layout. A unified CUDA kernel can then manage different LPs by computing direct memory offsets from the thread index, eliminating the overhead of indirect pointer access.

The second macro phase in the transformation focuses on synthesising CUDA code to invoke the behaviours declared by each actor, and thus associated with each logical process. The ActorLanguage assumes that actors' behaviours are expressed as sequences of ActorActions [11]. Each action can refer to implementation-specific statements (i.e., opaque behaviours) that support the execution of a step. Opaque behaviours usually depend on the specific target device; thus, in the IR, we assume that all these admissible variants are available to each actor. In this work, we express opaque behaviours as snippets in `mbeddr` [25].

An excerpt of the M2T transformation dealing with this aspect is reported in Figure 2. The M2T transforms each actor's behaviour into a collection of CUDA functions. For each message type, a dedicated function is generated to handle its reception. The `__device__` tag is prepended to indicate that the code must be executed on a GPU (line 4). The function signature is synthesised to allow the logical process that maps the considered actor to identify itself (i.e., the actor and the logical process in ROOT-Sim share the same ID). Also, the function accepts a generic state pointer (i.e., the actor's state),

```
1   /* GENERATION OF DEVICE FUNCTIONS (Event Handlers)
    ↪  */
2   foreach message in node.messages {
3     // Generate the function signature for the CUDA
      ↪  device
4     append {__device__ void process_event_}
      ↪  ${message.name} { (unsigned int me_idx,
      ↪  unsigned int lp_idx, (}
      ↪  ${node.actorCreation.ofConcept<ICreateActor>.
      ↪  first.stateType.name}
5        { *state) \{ } \n ;
6       ${node.actorCreation.stateType.
        ↪  getStructDeclaration().name}
7       { *me = (}
8       ${node.actorCreation.stateType.
        ↪  getStructDeclaration().name}
9       { *) state; } \n ;
10
11      // Inject the user-defined logic from the DSL
12      foreach stmt in message.body.statements {
13        // Translate DSL statements to CUDA
14        gen_statement(stmt);
15      }
16    append {\}} \n ;
17  }
```

Fig. 2: Excerpts of the M2T from ActorLanguage to CUDA.

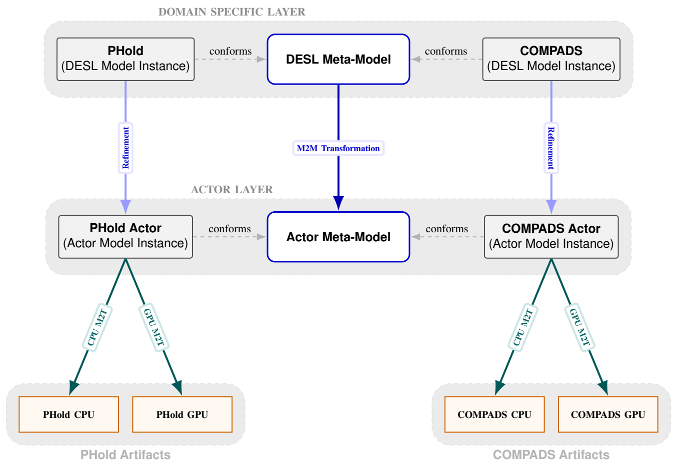

Fig. 3: A schematisation of the evaluation of the proposed software architecture.

which is immediately cast (lines 6–9) to a strongly typed C pointer corresponding to the specific actor's state.

The body of each function is completed with statements that reflect the ActorActions modelled in the Actor Model (lines 12–14). This is done by transforming `mbeddr` statements into concrete CUDA code that can implement the simulation state update.

Finally, the M2T transformation generates an event dispatcher that complies with the ROOT-Sim callback API. Indeed, when an event is scheduled for execution (corresponding to message extraction in the Actor Model), the RE invokes a single dispatcher provided by the model to activate the proper event handler, indicating the recipient LP. The generated dispatcher bridges the API of the selected RE with the actor model, resolving the architectural mismatch between the generic, type-agnostic event-delivery mechanisms of the underlying RE and the strongly typed, message-driven semantics of the intermediate representation.

## IV. Case Study

This section shows how, starting from the specifications of two different task-based applications modelled in DESL (see Section IV-A), it is possible to apply our M2M transformation to generate a corresponding representation in the Actor Model (see Section IV-B). Then, Section IV-C uses two M2T transformations to generate four different exercisable artefacts automatically. We show that these artefacts are actually executable on their respective devices (i.e., CPU, and GPU). The overall approach that we follow in this paper is depicted in Fig. 3.

### A. Benchmarks

For our case study, we have considered two benchmarks from the literature on Parallel Discrete Event Simulation (PDES), namely PHold [17] and COMPADS (COmparison of Parallel And Distributed Simulators) [18]. PHold is a well-established synthetic benchmark for evaluating the performance of PDES systems. Each LP processes an event by "holding" it for a certain amount of time (a computational delay) by relying on a busy loop. After the loop, the event is forwarded to another LP according to a stochastic distribution (e.g., uniform or selective, also called a hotspot). Our reference implementation of PHold in DESL is reported in Fig. 4.

Conversely, COMPADS is a benchmark suite conceived to fairly compare the performance of different PDES simulators in a reproducible manner. COMPADS includes a verification mechanism to ensure the correctness of the overall execution. Since the benchmark is deterministic, any correct implementation must produce the exact same sequence of events and final state hashes. This allows us to verify that our generated code is correct[1]. For space constraints, we do not report our COMPADS DESL code.

### B. The DESL Model and the M2M Transformation

DESL is a DSL rooted in Literate Programming and Model-Driven Engineering that allows modellers to specify PDES models in a high-level, declarative manner.

In our transformation pipeline, DESL abstracts the complexity of parallel execution by defining the simulation using two primary constructs: Logical Processes (LPs) and Events/Event Handlers. LPs represent the distinct entities within the simulation, encapsulating their own state and behaviour. The interactions between these entities are modelled exclusively through Events, which trigger the executions of LPs' events and move the simulation forward.

This structure aligns naturally with our intermediate representation. The concept of an LP maps directly to an actor, as both are autonomous units with a private state. Similarly, the event-driven mechanism of DESL, where LPs schedule and process events, corresponds to the message-passing paradigm of the actor model.

An excerpt of our M2M transformation is provided in Fig. 5. The first block iterates over event handlers to create the corresponding actor behaviours, using the `$$COPY_SRC$$` macro to directly transfer the transition logic from the simulation event handler to the actor's behaviour. This is possible exactly because we rely on `mbeddr` to represent the opaque behaviour

[1]The generated code was run on a RE already verified using COMPADS.

```
DES Model: Parallel Hold (PHold)

A synthetic benchmark maintaining a constant number
↪  of events in the simulation.

Events:

* EVENT

Constants:

The number of LPs in the model.
#define NUM_LPS = 262144;

Structs:

struct phold_msg {
   int dummy_data;
};

External Functions:

An external busy loop, for avoiding compiler
↪  optimizations.
void busy_loop(int max);

Configuration:

This is the number of iterations in the busy loop.
int loop_count = 100;
double mean = 1000.0;

Handlers:

Class classA:
 struct phold_state {
  int complete_events;
 };

 StartupFunction (int me, phold_state *state) {
    PHoldMessage new_event = {0};
    for (int32 i = 0; i < start_events; i++ ) {
        SendEvent EVENT to me at
        ↪  Exp(GetRandContext(), mean) with
        ↪  new_event
    }
 }

 handler EVENT (uint64 me, double now, uint32
 ↪  event_type, void* content, PHoldState* state) {
  busy_loop(loop_count);
  PHoldMessage new_event = {0};
  SendEvent EVENT to me at now + Exp(mean) with
  ↪  new_event
 }

Process Allocation:

assign_class([0, 262143], classA);
```

Fig. 4: Implementation of PHold in DESL.

```
$LOOP$ BEHAVIORS CreateBehavior($behavior, msg) {
   $LOOP$event HANDLER { $COPY_SRCL$  }

   No Become Statement
}

  [...]

$LOOP$CreateActors($<number>, $<baseName>,
↪  ->$[behavior], FIFO, ->$<stateType>);
```

Fig. 5: Excerpts of our M2M Transformation.

of the actors, thus easily preserving the original simulation semantics. The `No Become Statement` indicates that the actors do not change their behaviour after the processing of an event. This is expected because the logic of each LP remains static throughout the simulation, so must the behaviour of the actors.

The second block handles simulation deployment, instantiating parallel actors, binding them to the generated behaviours, and defining their memory layout. Finally, the injection of a FIFO queue guarantees the temporal ordering of events, a mandatory requirement for maintaining causality in the Time

```
static void PHoldClassA_behavior(lp_id_t me,
↪  simtime_t now, void *msg, struct phold_state
↪  *state) {
  busy_loop(loop_count);

  struct PHoldMessage new_event = { 0 };

  Envelope env = {
    .timestamp = now +
    ↪  cpu_random_exp(state->rng_state, mean),
    .sender = me,
    .receiver = me,
    .type = EVENT
  };
  new_event = {
    .envelope = env,
    .payload = new_event
  };

  ScheduleNewEvent(env.destination, env.timestamp,
  ↪  env.type, &new_event, sizeof(Message));
}
```

Fig. 6: Excerpt of the Generated CPU Code for PHold.

```
__device__
char PHoldClassA_behavior(uint64_t me, double now,
↪  Event *msg) {

  curandState_t *cr_state = &(nodes.cr_state[me]);

  /* save state before updates */
  State old_state = {
    .cr_state = nodes.cr_state[me],
  };

  busy_loop(loop_count);

  struct Event new_event = { 0 };

  Envelope env = {
    .timestamp = now +
    ↪  gpu_random_exp(state->rng_state, mean),
    .sender = me,
    .receiver = me,
    .type = EVENT
  };
  new_event = {
    .envelope = env,
    .payload = new_event
  };

  char append_res =
  ↪  append_event_to_queue(&new_event);
  if (append_res == 0) {
    nodes.cr_state[me] = old_state.cr_state;
    return 11;
  }

  return 1;
}
```

Fig. 7: Excerpt of the Generated GPU Code for PHold.

Warp optimistic synchronisation protocol managed by the underlying heterogeneous runtime.

### *C. M2T Transformation and Execution Results*

For the first part of our experimental assessment, we have generated C and CUDA versions of the PHold models. In Fig. 6 and Fig. 7 we report the generated code for the CPU and GPU versions, respectively. As can be seen, the simulation's core behaviour is identical in both listings. The logic for processing the event and choosing the next destination remains unchanged, but differences in the underlying architectures can be observed in how CUDA kernels use device functions and random number generators, as well as in the specific API used to interact with the RE when running on CPUs or GPUs.

The distribution of events in this PHold implementation is mostly uniform: each LP schedules the event to another LP chosen uniformly at random. We then modified the original

```
1   handler EVENT (uint64 me, double now, uint32
    ↪  event_type, void* content, PHoldState* state) {
2     busy_loop(loop_count);
3
4     lp_id_t dest = me;
5     if (Random() < 0.30) {
6       dest = ((lp_id_t)(Random() * NUM_LPS))
7     }
8
9     PHoldMessage new_event = {0};
10    SendEvent EVENT to dest at now + Expent(mean) +
      ↪  lookahead with new_event
11  }
```

Fig. 8: Change Introduced in PHold DESL.

```
static void phold(lp_id_t me, simtime_t now, void
↪  *msg, struct phold_state *state) {
  busy_loop(loop_count);

  lp_id_t dest = me;
  if (cpu_random(state->rng_state) < 0.30) {
    dest = ((lp_id_t)((cpu_random(state->rng_state)
    ↪  * NUM_LPS)));
  }

  struct PHoldMessage new_event = { 0 };

  Envelope env = {
    .timestamp = now +
    ↪  cpu_random_exp(state->rng_state, mean),
    .sender = me,
    .receiver = me,
    .type = EVENT
  };
  new_event = {
    .envelope = env,
    .payload = new_event
  };

  ScheduleNewEvent(env.destination, env.timestamp,
  ↪  env.type, &new_event, sizeof(Message));
}
```

Fig. 9: Excerpt of the Generated CPU Code after Change.

DESL code to implement an unbalanced distribution, allocating 70% of events to a subset of the LPs, creating an imbalance in the workload. The modified part of the DESL code is reported in Fig. 8 (lines 5–8) .

We then reran both our M2M and M2T transformations, obtaining the changes reported in Fig. 9 and Fig. 10. As expected, the new logic is present in both outputs. The C and CUDA code versions now include the conditional statements to direct 70% of the events to the target subset. This confirms that the architecture successfully propagates the functional change across all distinct physical implementations, ensuring that CPU and GPU simulations remain semantically identical and eliminating the risk of human error during updates.

We have run the generated models to assess the performance and correctness of our generated code. We have used a machine equipped with an AMD Ryzen 5 3600 CPU (12 vCPUs in total) and 16 GB of RAM, and with an Nvidia RTX 2070 GPU with 8 GB of RAM. Regarding PHold, we report the results in Table I. As mentioned, we have used ROOT-Sim as the main RE, which embodies the traditional Time Warp algorithm [26] for CPUs and the GPU-TW algorithm [27] for GPUs, and enables co-execution across heterogeneous devices using the Follow The Leader [28] execution scheme. Thanks to this RE, we have therefore compared the performance of the generated code for both the balanced (Fig. 4 ) and the unbalanced (Fig. 8) versions. In these experiments, we have

```
__device__
char PHoldClassA_behavior(uint64_t me, double now,
↪  Event *msg) {

  curandState_t *cr_state = &(nodes.cr_state[me]);

  /* save state before updates */
  State old_state = {
    .cr_state = nodes.cr_state[me],
  };

  busy_loop(loop_count);

  lp_id_t dest = me;
  if (random(cr_state, g_n_nodes) <= 0.30)
  {
    dest = ((lp_id_t)((random(cr_state, g_n_nodes))
    ↪  * NUM_LPS));
  }

  struct Event new_event = { 0 };

  Envelope env = {
    .timestamp = now +
    ↪  gpu_random_exp(state->rng_state, mean),
    .sender = me,
    .receiver = me,
    .type = EVENT
  };
  new_event = {
    .envelope = env,
    .payload = new_event
  };

  char append_res = append_event_to_queue(&new_msg);
  if (append_res == 0) {
    nodes.cr_state[me] = old_state.cr_state;
    return 11;
  }

  return 1;
}
```

Fig. 10: Excerpt of the Generated GPU Code after Change.

| Configuration | CPU | GPU | Co-Execution |
|---|---|---|---|
| *balanced* | 138.132 | 18.614 | 32.569 |
| *unbalanced* | 124.936 | 63.049 | 110.500 |

TABLE I: Performance Results (in seconds)

used 262,144 LPs and a loop count of 100.

As shown in the results, the GPU implementation excels in the balanced configuration, outperforming the CPU by approximately 18 seconds. However, introducing an unbalanced workload increases GPU execution time to about 63 seconds. This aligns with the known limitations of SIMT (i.e., Single instruction, multiple threads) hardware architectures, which suffer when parallel threads have divergent workloads. In contrast, the CPU performance remains consistent across both scenarios. The co-execution strategy falls between the two extremes but is similarly impacted by the load imbalance. Ultimately, these distinct performance behaviours validate our generation process, demonstrating that the automatically synthesised code correctly exhibits the native performance characteristics of each target hardware platform. Moreover, these results are consistent with what was already observed in [28]. Nevertheless, unlike [28], no manual implementation of the code for CPUs and GPUs was required in this experiment.

Finally, we have run COMPADS to verify the correctness of the generated code. As mentioned, the model computes a sequence of checksums deterministically across the simulation for each LP. The simulation is considered correct if, for each LP, all checksums match the expected ones. In Table II

| Event # | CPU checksum | GPU checksum |
|---|---|---|
| 1 | 3251074139 | 3251074139 |
| 2 | 2954563034 | 2954563034 |
| 3 | 554745614 | 554745614 |
| 4 | 942223923 | 942223923 |
| 5 | 885309220 | 885309220 |
| 6 | 982526139 | 982526139 |
| 7 | 878741649 | 878741649 |

TABLE II: Final Sequence of Checksums for one LP

we report the checksums for the last 7 events in the chain, computed by LP 0, for both the CPU and GPU versions of the code. As can be seen, the checksums generated by the CPU execution match perfectly with those produced by the GPU. This is a critical result. Even though the underlying hardware architectures use vastly different mechanisms for memory and arithmetic, the sequence of computed states remains identical. This confirms that our automated transformation pipeline preserves the deterministic semantics of the original model, producing different functionally equivalent versions.

## V. Related Work

General-purpose heterogeneous programming frameworks, including OpenCL [29], SyCL [30], and Kokkos [31], abstract underlying hardware resources but lack the domain-specific primitives necessary for expressive application formulation. These approaches concentrate exclusively on the software development lifecycle, omitting explicit runtime support for the execution and optimal scheduling of software bundles across diverse hardware topologies. The architecture we have expanded in this paper resolves the specification and runtime orchestration through a comprehensive MDE pipeline. In particular, with the proposal in this paper, we decouple the domain logic from the physical hardware, automating also the generation of CUDA targets and their dynamic orchestration without manual intervention.

Model-Driven Engineering methodologies targeting heterogeneous systems have gained significant momentum in the literature, presenting various abstraction paradigms to mitigate underlying hardware complexity. In [32], the authors propose ThingML, a DSL to synthesise applications for diverse targets, from microcontrollers to servers. In [33], the author proposes a model compiler that automatically infers sequential, data-parallel, or task-parallel execution semantics, eliminating the manual burden of writing resource-specific offloading and communication routines for hybrid hardware. In [34], the authors explicitly target PDES execution on heterogeneous systems using MDE-generated assets, similarly to what we discussed in our use case. We share the fundamental goals and objectives of all these works, but our architecture is fundamentally different. We introduce a platform-independent Actor Model as a central intermediate layer. This separation means we only need to develop the M2T code generator exactly once. Because this M2T phase translates directly from the intermediate layer, the system targets various hardware classes. Furthermore, this design makes adding new DSLs very efficient. Developers simply create a single M2M transformation to convert the new DSL into the Actor Model. Instantly, applications written in that new language can run on any supported heterogeneous accelerator, entirely reusing the existing M2T generators without any modifications.

## VI. Conclusions and Future Work

In this paper, we presented a software architecture that decouples logical specifications from physical hardware implementations in applications for Heterogeneous Computing environments. The proposed software architecture leverages MDE technologies to transform applications expressed in DSLs into a unified IR based on the Actor Model; it also includes specific assets to automatically synthesise IR artefacts into C and CUDA executables.

Evaluations using PDES benchmarks confirmed that the proposed software architecture guarantees functional equivalence and automates change propagation across heterogeneous targets without degrading performance. Future work will evaluate the entire Domain stack by translating applications from diverse DSLs into the Actor Model via dedicated M2M transformations. This will demonstrate the architectural interoperability and extensive reusability of the existing M2T assets for heterogeneous code generation.

## Acknowledgment

This paper has been partially supported by European Union—Next Generation EU, Mission 4, Component 2, CUP E53D23008200006 (Domain). Guglielmo De Angelis is with the Italian Research Group: INdAM-GNCS.

## References

[1] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang, "Heterogeneous computing: challenges and opportunities," *Computer*, vol. 26, no. 6, pp. 18–27, Jun. 1993.

[2] M. Zahran, "Heterogeneous computing: here to stay," *Communications of the ACM*, vol. 60, no. 3, pp. 42–45, Feb. 2017.

[3] J. Dongarra, P. Beckman, T. Moore *et al.*, "The international exascale software project roadmap," *The international journal of high performance computing applications*, vol. 25, no. 1, pp. 3–60, 2011.

[4] J. Dongarra, S. Gottlieb, and W. T. C. Kramer, "Race to exascale," *Computing in science & engineering*, vol. 21, no. 1, pp. 4–5, Jan. 2019.

[5] D. M. Kunzman and L. V. Kale, "Programming heterogeneous systems," in *Int. Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, May 2011, pp. 2061–2064.

[6] O. Terzo, K. Djemame, A. Scionti, and C. Pezuela, Eds., *Heterogeneous computing architectures: Challenges and vision*. London, England: CRC Press, Sep. 2019.

[7] J. Fang, C. Huang, T. Tang, and Z. Wang, "Parallel programming models for heterogeneous many-cores: a comprehensive survey," *CCF Trans. on High Performance Computing*, vol. 2, no. 4, pp. 382–400, Dec. 2020.

[8] S. Jha, M. Cole, D. S. Katz *et al.*, "Distributed computing practice for large-scale science and engineering

applications," *Concurrency and computation: practice & experience*, vol. 25, no. 11, pp. 1559–1585, Aug. 2013.

[9] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, "Event-driven programming for robust software," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC - EW10*. New York, New York, USA: ACM Press, 2002.

[10] A. K. Sujeeth, K. J. Brown, H. Lee *et al.*, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. on Embedded Computing Systems*, vol. 13, no. 4s, pp. 1–25, Apr. 2014.

[11] S. Bauco, G. De Angelis, R. Marotta, and A. Pellegrini, "A model-driven platform for software applications on heterogeneous computing environments," in *Proceedings of the 22nd IEEE International Conference on Software Architecture Companion*, ser. ICSA'25. Piscataway, NJ, USA: IEEE, Mar. 2025.

[12] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice*, ser. Synthesis lectures on software engineering. Cham: Springer International Publishing, 2017.

[13] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," *International Joint Conference on Artificial Intelligence*, pp. 235–245, Aug. 1973.

[14] G. Agha, "An overview of actor languages," *SIGPLAN notices*, vol. 21, no. 10, pp. 58–67, Oct. 1986.

[15] P. Wilsdorf, A. Wolpers, J. Hilton, F. Haack, and A. M. Uhrmacher, "Automatic reuse, adaption, and execution of simulation experiments via provenance patterns," *ACM Transactions on Modeling and Computer Simulation*, vol. 33, no. 1-2, pp. 1–27, Sep. 2022.

[16] S. Bauco, R. Marotta, and A. Pellegrini, "DESL: A literate programming language framework for interoperable parallel discrete event simulation," in *Proceedings of the 2025 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '25. New York, NY, USA: ACM, Jun. 2025, p. 12.

[17] R. M. Fujimoto, "Performance of time warp under synthetic workloads," in *Distributed Simulation*, ser. PADS'90, D. Nicol, Ed. San Diego, CA, USA: Society for Computer Simulation International, 1990, pp. 23–28.

[18] T. Köster, A. M. Uhrmacher, and P. Andelfinger, "Towards an open repository for reproducible performance comparison of parallel and distributed discrete-event simulators," in *Proc. of ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. New York, NY, USA: ACM, Jun. 2022, pp. 31–32.

[19] G. Agha, "An overview of actor languages," *ACM Sigplan Notices*, vol. 21, no. 10, pp. 58–67, 1986.

[20] S. Conoci, P. Di Sanzo, A. Pellegrini, B. Ciciani, and F. Quaglia, "On power capping and performance optimization of multithreaded applications," 2021.

[21] E. De Angelis, G. De Angelis, R. Marotta *et al.*, "Declarative adaptive optimization of task-based applications on heterogeneous architectures," in *Proc. of IEEE/SBC 37th Int. Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, 2025.

[22] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance," *Commun. ACM*, vol. 54, no. 12, p. 92–103, Dec. 2011.

[23] V. Pech, "JetBrains MPS: Why modern language workbenches matter," in *Domain-Specific Languages in Practice*. Cham: Springer International Publishing, 2021, pp. 1–22.

[24] A. Pellegrini, R. Vitali, and F. Quaglia, "The ROme OpTimistic simulator: Core internals and programming model," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTOOLS. Brussels, Belgium: ICST, Apr. 2012, pp. 96–98.

[25] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "mbeddr: an extensible C-based programming language and IDE for embedded systems," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, ser. SPLASH '12. New York, NY, USA: ACM, Oct. 2012, pp. 121–140.

[26] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[27] X. Liu and P. Andelfinger, "Time warp on the GPU: Design and assessment," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '17. New York, NY, USA: ACM, May 2017, pp. 109–120.

[28] R. Marotta, A. Pellegrini, and P. Andelfinger, "Follow the leader: Alternating CPU/GPU computations in PDES," in *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS'24. New York, NY, USA: ACM, Jun. 2024, pp. 47–51.

[29] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, vol. 12, pp. 66–72, 2010.

[30] T. K. G. Inc, "Sycl™ 2020 specification (revision 9)," 2020, accessed: 2024-10-17.

[31] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE, Aug. 2013.

[32] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, "ThingML: a language and code generation framework for heterogeneous targets," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. New York, NY, USA: ACM, Oct. 2016.

[33] F. Ciccozzi, "Towards accessible software engineering for heterogeneous hardware," in *International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA)*. IEEE, Feb. 2024, pp. 1–6.

[34] R. Marotta and A. Pellegrini, "Model-driven engineering for high-performance parallel discrete event simulations on heterogeneous architectures," in *2024 Winter Simulation Conference (WSC)*. IEEE, Dec. 2024, pp. 2202–2213.