DESL: A Literate Programming Language Framework for Interoperable Parallel Discrete Event Simulation

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

Romolo Marotta
r.marotta@ing.uniroma2.it
Tor Vergata University of Rome
Rome, Italy

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

Abstract

Simulation is indispensable for advanced scientific research, enabling accurate explorations of complex phenomena and supporting evidence-based decision-making across interdisciplinary boundaries. Parallel Discrete Event Simulation (PDES) provides substantial advantages in modelling large-scale systems by distributing computational tasks among multiple processors, enhancing scalability. However, exploiting it is extremely challenging due to obstacles in model efficiency, concurrency control, reproducibility, and maintainability. Furthermore, the large number of available PDES runtime environments makes it difficult to explore their (performance) capabilities for some specific model, hindering the identification of the best-suited technology for a certain simulation study. To address these limitations, we introduce a unified framework grounded in literate programming and model-driven engineering, integrating interwoven documentation and model logic within a single source. This design enhances intrinsic consistency between model logic and explanatory content, while enabling the generation of model implementations tailored to multiple runtime environments, thus allowing simulationists to focus on model development without being locked in to any specific technology or environment. This facilitates model reuse and performance comparisons across diverse execution environments. We show the viability of this approach by providing the first-ever experimental comparison across three different simulators, starting from the same model implementation.

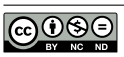
CCS Concepts

• **Computing methodologies** → **Discrete-event simulation**; *Simulation theory*; • **Software and its engineering** → **Model-driven software engineering**.

Keywords

Parallel Discrete Event Simulation, Model Driven Engineering

ACM Reference Format:
Simone Bauco, Romolo Marotta, and Alessandro Pellegrini. 2025. DESL: A Literate Programming Language Framework for Interoperable Parallel Discrete Event Simulation. In *39th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '25), June 23–26, 2025, Santa Fe, NM, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3726301.3728420



SIGSIM-PADS '25, Santa Fe, NM, USA

ACM ISBN 979-8-4007-1591-4/2025/06
https://doi.org/10.1145/3726301.3728420

1 Introduction

Simulation is a fundamental tool for science, facilitating a deeper understanding of complex systems and enabling informed decision-making in diverse domains. Parallel Discrete Event Simulation (PDES) [14] is considered one of the most effective methodologies for simulating large-scale systems, offering significant scalability by distributing the simulation workload across multiple processors. PDES leverages parallelism to reduce execution time, making it particularly suitable for domains where the complexity of the systems under study demands competitive simulation times.

Despite its importance, the development, maintenance, and execution of (P)DES models[1] often face challenges that hinder their effectiveness. Key among these challenges is the *fragmentation of simulation workflows*, where documentation and implementation are treated as distinct entities, leading to inconsistencies and difficulties in maintaining models [43]. Furthermore, the multiplicity of PDES simulation runtime environments (REs) exacerbates the complexity of adapting a single model for performance evaluation across different platforms, thus limiting comparative analyses and the selection of the best-suited capabilities available in off-the-shelf simulators [8].

Literate programming [25] has long emphasised the importance of integrating documentation and code to produce software that is not only functional but also comprehensible. This paradigm promotes the view of programs as works of literature, prioritising human understanding alongside computational correctness. At the same time, model-driven engineering (MDE) [5] techniques have provided systematic methodologies to develop and transform MODELS at different levels of abstraction, ensuring consistency and traceability throughout the development lifecycle. However, the integration of these paradigms into the specific context of model development and execution remains fairly underexplored, although with few exceptions (see, e.g., [29, 49, 53]).

In this paper, we introduce the Discrete Event Simulation Language (DESL, pronounced[2] /'di:zəl/) framework, a *literate programming language* framework for PDES. DESL leverages MDE to extend the principles of literate programming to the domain of simulation, providing both a Domain-Specific Modelling Language

[1] Throughout this paper, we face the need to distinguish the dual sense of the term "model". We use a different typographical notation for this purpose. Whenever we refer to the *conceptual (meta-)model* proper of MDE techniques, we use the small caps typeface MODEL. When referring to a *simulation model* as the representation of the system or process under study, we use the sans-serif typeface model.

[2] The initial concept for the DESL framework originated approximately 12 years ago. During this period, DESL work has been initiated at least six times, using different tools and methods, each attempt culminating in outcomes deemed insufficiently satisfactory and ultimately discarded. The name "DESL" and its pronunciation playfully allude to the iterative nature of its development, akin to the modest acceleration of a diesel engine. However, once the system reaches its steady state, it delivers significantly greater "horsepower" in the form of robust functionality and performance.

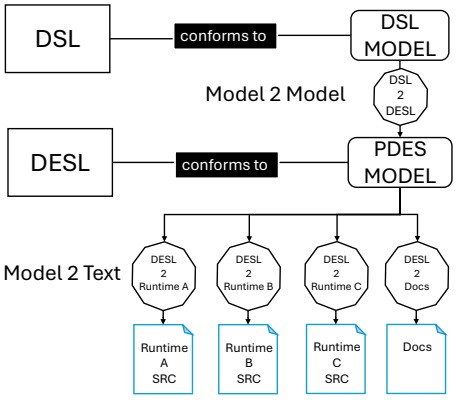


Figure 1: The DESL Workflow.

(DSML) to implement platform-independent models and a Model-Driven Architecture to generate platform-specific artefacts that can be executed on different REs. It enables the creation of models in which documentation and code are interwoven within a single source. From this unified source, DESL facilitates the generation of simulation code that can be executed across multiple REs, thereby addressing two critical needs: ensuring that documentation and code are inherently kept consistent, and providing a mechanism for comparative performance evaluation of models across diverse platforms. DESL is a framework in the sense that targeting a new or different RE is trivial, requiring only to plug in a new MODEL *transformation* [32], allowing all models developed in DESL can be executed on the new RE. DESL is also a framework because, being based on a formal MODEL, Domain-Specific Languages (DSLs) could be transformed to DESL, thus benefitting from the capability to run on different REs. Overall, with DESL, we envision the workflow depicted in Figure 1, in compliance with the vision in [53].

DESL operates on a platform-independent MODEL that captures the semantics and structure of the simulation. This MODEL enables us to derive executable artefacts and human-readable documentation automatically, by providing the essential elements to describe generic DES models. By employing MODEL transformations, DESL provides extensibility to accommodate the technical requirements to execute the final model on different REs. The benefit is that, if MODEL transformations are formally verified, the resulting model can be deemed correct, and comparative execution across different REs can be supported with no manual intervention.

The ability to execute the same model across different runtime environments (REs) is essential for evaluating model's relative performance and understanding the trade-offs inherent in each platform. Indeed, PDES is an umbrella term: there are multiple synchronization algorithms [37] (ranging from completely conservative [10] to fully optimistic [21], with many intermediate flavours [1, 22, 40, 41]), which can be executed on shared memory systems (see, e.g., [11, 19, 42, 45, 47, 48]), in entirely distributed settings (see, e.g., [9, 31, 36]), using a mixture of checkpointing-based protocols or reverse computation protocols (see, e.g., [9, 12, 23]), just to mention a few attributes that characterize existing REs. The literature has firmly shown (see, e.g., [2, 30, 37]) that there is no silver bullet, and different workload profiles can benefit from any combination of these solutions: selecting a RE is therefore crucial for the performance of simulation studies.

Unfortunately, the interaction dynamics between the model and the RE only become apparent *after* the model is implemented and (preliminary) executions are performed. At this stage, changing the technology involved may be too late: different REs typically also support different programming languages and programming models. The cost of switching could be too high. Hence, simulation studies typically risk to be performance ineffective, due to a selection of a technology in a too-early stage of development, which could become a lock-in for time and budget reasons.

The DESL framework targets this *interoperability wall* exactly thanks to the MDE capabilities discussed above. A model coded in DESL will benefit from multiple MODEL transformations that can deliver concrete implementations targeting different REs with no need to touch the code written by modellists. In this way, multiple REs, algorithms, and methodologies could be compared out of the box, allowing model developers to select the best-suited technology for their study, and RE developers to optimise their implementations under real workloads. This scenario has long been sought by the simulation community [18, 53] and, to the best of our knowledge, we are the first ones practically realizing a solution relying on MDE allowing to compare off-the-shelf REs.

Incidentally, thanks to this work, we highlight that if the source code is considered a MODEL rather than a textual artefact, the distinction between *program generation* and *documentation generation* proper of the literate programming paradigm is actually subsumed to the single notion of *MODEL transformation*. This result, even though immediate and well accepted in the software engineering community, could benefit also the simulation community, which is currently highlighting the need for consistent documentation of the models also during their development (see, e.g., [17]).

Overall, this paper makes the following contributions: ① it introduces a framework that integrates literate programming and model-driven engineering to address the need for consistent documentation aligned with the evolution of the models; ② it demonstrates how a single code base can serve as the basis for generating both documentation and executable artefacts; ③ it showcases a methodology and presents an implementation of it to allow for the comparison of the capabilities of multiple REs starting from the same code base, highlighting the benefits of cross-platform simulation execution.

We release DESL as open-source software[3], also in the hope that the developers of DES runtime environments will consider the implementation of components to plug their environments, fostering increased interoperability among DES modelists.

The remainder of this paper is structured as follows. In Section 2 we discuss prior work. The DESL Framework is presented in Section 3. A preliminary experimental assessment, showing the capability to execute on three REs taken from the simulation community, is provided in Section 4. Some threats to the validity of these experimental evaluations are discussed in Section 5.

[3]The DESL living repository can be found at https://github.com/DomainProject/DESL. The persistent version associated with this publication is available at https://doi.org/10.5281/zenodo.15298361.

2 Related Work

Generating code from higher-level specifications is not a new idea, and there are several proposals that have aimed at this goal.

In [39], the authors explore natural language processing as a way to build a model specification from a narrative that describes the functioning of the system under study. While the proposal effectively allows for deriving essential elements of the model, it does not allow for automatically generating the entire model artefact. We are able to produce working models because we do not start from ambiguous natural language descriptions: DESL grounds on a formal MODEL, which can be unambiguously translated into some program, while providing a higher level of abstraction. The capability of interweaving documentation and code permits to keep the "narrative" part of the work in [39] within the same code base.

Generating working models (semi) automatically has been explored in [27, 33]. Particularly, the work in [33] has focused on the generation of Spiking Neural Network (SNN) models for heterogeneous architectures. The idea behind the work is that, in SNN models, a large part of the code is typically the same, while some aspects the neuron/synapse specification and their interconnections are the tiny parts that change. Using the concept of *templates* and relying on LLVM, the authors can generate parts of the model that are then plugged into an optimised RE based on NEST [16]. Conversely, in [27], the authors rely on code generation interweaved with optimization techniques to generate, starting from DSLs in the realm of reaction networks, a simulator that is optimised for the specific model. The main difference with our work, in both cases, is that we do not target a specific domain—DESL is not a DSL—but rather we aim at interoperability between different REs.

Comparing the performance of different REs is something that has been repeatedly sought in the speculative DES literature. To this end, many benchmarks have been proposed, such as PHold [15] (and multiple variants, like EPHold [4] or Mem-PHold [44]), LaPDES [34], and COMPADS [26]. While the first two are mainly targeted at performance evaluation, the latter also takes into account the correctness of the implementation of the RE. All these benchmarks have two characteristics in common: they are synthetic, and they lack a standardization. In particular, in order to use these benchmarks, RE developers have to re-implement their own version—a notable exception is [26], where the authors of the benchmarks have provided implementations for multiple REs. These two characteristics can be problematic for a fair comparison: first, different implementations may not necessarily be equivalent; second, a synthetic workload may be non-representative of real-world models. DESL paves the way to the possibility of having a benchmark suite of real-world models that, if the transformations to RE code are formally verified as equivalent, will provide out of the box a reliable comparison.

One feature of DESL is that, despite targeting PDES, it does not expose simulation practitioners to the details of the REs, of the synchronization protocol enforced by the REs, or of the memory management capabilities of the RE (such as state saving or reverse computation). This feature has already been explored in APOSTLE [7], a (not so domain-specific) DSL for PDES. The main difference with DESL is that we can generate concrete implementations of the models targeting multiple REs, enabling performance comparisons. Additionally, APOSTLE has no notion of literate programming as DESL does.

Another high-level model description is the Discrete Event System Specification (DEVS) [52]. 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. DESL is not based on a mathematical foundation, but rather on a MODEL that can be automatically transformed into concrete implementations targeting multiple REs. In this sense, DESL and DEVS are completely orthogonal approaches to model specification.

Several works have applied Model-Driven Engineering (MDE) to automate model generation. For instance, in [29] the authors propose an MDE model automatically transformed into versions suitable for heterogeneous architectures. However, their approach differs from ours, as DESL explicitly targets interoperability, enabling practitioners to execute identical models across multiple REs. The work in [53] advocates an MDE-based domain-specific modeling approach focused on trustworthy agent-based simulations by improving traceability and reproducibility, particularly for critical decision-making contexts. Unlike this structured modeling perspective, DESL emphasizes platform-agnostic execution and rigorous performance benchmarking—an aspect absent in Zschaler et al. Additionally, DESL incorporates literate programming to unify logic and documentation, a feature lacking in both referenced works.

Another relevant work that emphasizes the importance of model documentation across the lifetime of model development and simulation studies is [50]. The authors introduce a framework for the automatic reuse, adaptation, and execution of simulation experiments using provenance patterns, and address the challenge of systematically conducting simulation studies by leveraging provenance graphs that document the history of modelling and experimentation activities. Their approach identifies reusable simulation experiments, adapts them to new scenarios, and executes them automatically, reducing manual intervention. At the same time, this work does not consider the opportunity of comparing the performance of real-world models on multiple REs, as we do. In this sense, these two works can be considered orthogonal, and DESL may benefit also the research line depicted in [50].

Interoperability between models is traditionally addressed by the High Level Architecture (HLA) [20], a standard facilitating coherent data exchange and synchronized time management across simulators in distributed federations. Unlike HLA, our work pursues a different form of interoperability, focusing on executing the same model across multiple REs, a capability HLA lacks since it relies on manually implemented federates tailored to each RE.

OpenABL is a domain-specific language for agent-based simulations capable of generating code for multiple simulation platforms, such as Flame GPU, Repast, and Mason [13]. However, its traditional grammar-based approach limits extensibility, complicating both DSL syntax expansion and the integration of new simulation REs. This rigidity motivated our shift toward a more modular Model-Driven Engineering approach with DESL. The same considerations hold for other grammar-based approaches such as [3].

3 The DESL Framework

The DESL Framework is based on a set of software artefacts that allow developers to implement models in a succinct way, and delegate to *transformations* the burden of providing a concrete representation of their model, which can be compiled to a computer program and linked against some specific RE. DESL embeds *literate programming* concepts, allowing the documentation of the model to be interweaved with the model's logic, thus enforcing maintainable documentation throughout the simulation lifecycle.

DESL employs Model-Driven Engineering (MDE) techniques to automatically generate executable code compatible with multiple runtime environments (REs). MDE is a software development paradigm centred around formal MODELS, which serve as core artifacts enabling abstraction, automation, and streamlined system design. Fundamental to this approach are model-to-model (M2M) transformations, allowing MODELS to be systematically refined or optimized, promoting interoperability between different abstraction layers and languages. Complementing M2M transformations, model-to-text (M2T) transformations translate platform-independent MODELS directly into concrete source code, configurations, or documentation, thus facilitating a reliable and automated transition from high-level specifications to executable artefacts.

Literate programming generates documentation and code relying on two *processors*, named *tangle* and *weave*: the former extracts the actual program to be compiled; the latter generates a markup document that can be then used to generate the actual documentation—in the original work [25] it was meant to be TEX source code. From an MDE perspective, the tangle and weave "processors" can be seen as two different M2T transformations. Tangle generates compilable source code, while weave produces some formatted document that interleaves commentary with code.

An additional advantage of DESL's MDE approach is the possibility to leverage M2M transformations. Since DESL grounds on a MODEL, it becomes feasible to define M2M transformations that map various MODELS to DESL programs, implemented in the native DESL DSML. This enables the development of multiple DSLs tailored to specific modelling scenarios while seamlessly leveraging DESL's interoperability features. When a DSL is translated into a DESL program via an M2M transformation, all the existing M2T transformations provided by DESL become automatically available. As a result, models expressed in the original DSLs can be executed on any runtime environment supported by DESL without requiring additional transformation efforts.

This section details the DESL MODEL and the capabilities of the DESL framework, first illustrating the MODEL itself, and then presenting the M2T transformations that we have implemented to showcase the capabilities of our MDE approach.

3.1 The Core MODEL

We have realised the DESL MODEL using the JetBrains Meta Programming System (MPS) as the reference Language Workbench [35]. The DESL MODEL is depicted in Figure 2, where the fundamental concepts and their relations are reported.

The central concept is the `DES Model`. This concept encapsulates an entire DES model implemented with DESL, which we call a *DESL program*. To support flexibility in the development, a DESL program is organised into sections, each of which can be used to specify different parts of the model.

Typically, DES models can be seen as composed of two main parts. The first part is the *model configuration*, where the initial setup of the model execution is carried out. This aspect of modelling is captured by the global `StartupFunction`. A `StartupFunction` is a procedural intializationation of the model, that is run before the actual simulation is started. In the `StartupFunction`, for example, command line arguments can be processed, configuration files can be loaded, and in general any activity related to the setup of a single execution of the model can be carried out. For generality, the global `StartupFunction` is optional.

The second essential part of a DESL program is the encoding of the logic of the model. According to traditional DES models organization, this logic is delegated to a set of *event handlers*. Typically (and with very few exceptions, such as [23]), event handlers are in charge of executing a single event targeted to some part of the model—borrowing from the traditional PDES literature [14], we refer to such parts of the models as *Logical Processes* (LPs). We therefore capture this behaviour of event handlers with the `EventHandler` concept. In a DESL program, any number of `EventHandlers` can be specified. The specific type of event that each handler is in charge of execution is defined through the `EventDefinition`, which allows mapping one event to a string literal (its name). Event definitions appear in a dedicated section of a DESL program.

The logic of event handlers can be arbitrary. Each handler is associated with exactly one `Function`, which allows to specify the arbitrary logic. In DESL, these functions can be of two different types. The principal type (described by the `Function` class itself) is a function directly encoded in a DESL program. For this purpose, we rely on the mbeddr [46] MODEL to allow the specification of the model logic directly in a DESL program. In this way, DESL event handlers can rely on a C-like syntax to manage state variables, operators, and function calls. Therefore, a `Function` is a sequence of `Statements` extending mbeddr statements.

For model development convenience, in DESL it is also possible to perform function calls to any external library function[4]. This concept is again captured by the `Function` concept. Indeed, such function calls can be linked to `ExternalFunctions`, which are described by an `ExternalFunctionPrototype` that can appear in a dedicated section of a DESL program.

One important abstraction that we have introduced in DESL is the concept of `ClassDefinition`. This concept captures the recurring notion in DES model of *classes of LPs*. Indeed, in a model, we can have a disparity of different elements that behave differently and respond differently to events. As an example, consider Agent-Based model: in this domain, a single model could encompass different types of agents, each having their own behaviour. Agents belonging to different classes of agents could respond differently to the same event generated, e.g., from the environment. It is therefore convenient to encode the logic of the different agents independently. A `ClassDefinition` allows to group together different

[4] Clearly, if the model will be executed on an optimistic RE, such functions must either be stateless, or they must be rollback-aware: this is some complication that DESL does not target to solve, also because there are already some proposals in the literature that have attempted this pathway [12, 38].

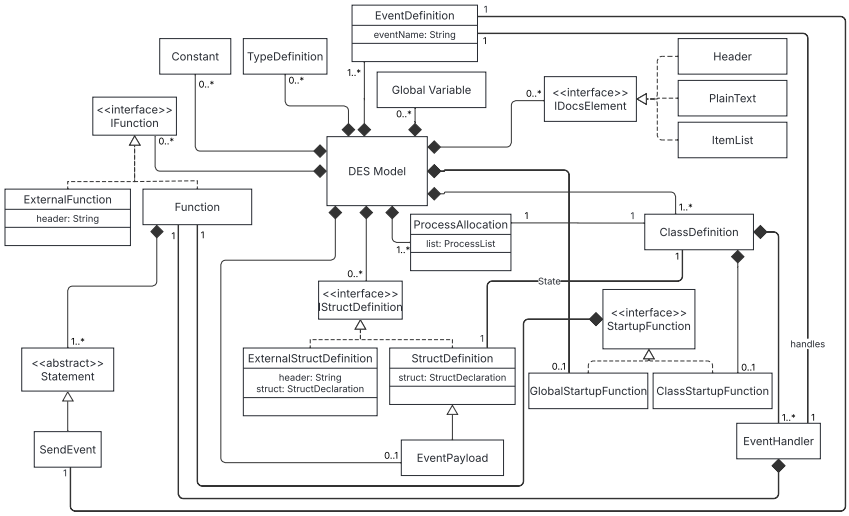


Figure 2: The DESL MODEL. For simplicity, we do not show relations with mbeddr concepts, and some attributes are omitted.

`EventHandlers` that jointly implement the behaviour of a homogeneous class of LPs. Each class can have its own `StartupFunction`, which we discuss later. When the model is deployed and executed, simulationists are typically interested in having *populations* of different classes of LPs behaving uniformly. This idea is captured by the `ProcessAllocation` concept, which allows for the exact construction at model startup of such populations.

A key concept when developing DES models is the ability to exchange events between different LPs. Typically, different REs offer different supports to inject events into the model. We have therefore abstracted away the event scheduling functionality by using the `SendEvent` concept. `SendEvent` is a `Statement`, and can therefore appear in any point of a `Function` associated with an `EventHandler`. When injecting a new event through the `SendEvent` concept, the modeller can specify what `EventDefinition` is involved (using its string literal) and what LP (defined in the aforementioned `ProcessAllocation` concept) is the recipient of such event—when selecting a target LP, developers are also provided with additional advanced capabilities that are discussed in Section 3.2.2. Additionally, the modeller can specify a timestamp (a simple real value) at which the event should be scheduled, and the *payload* of the event, i.e. any data structure that is piggybacked by the event.

Data structures are another fundamental component of DESL. DESL is a *strong-typed language*, thanks to the `TypeDefinition` concept. This concept leverages the capabilities of mbedder to declare structured data types (resembling C-like `structs`), which are mapped to the `StructDefinition` concept in DESL. Such datatypes are used to define what is the payload of events injected using the `SendEvent` concept.

Another use of the `StructDefinition` in DESL is the specification of the simulation state of LPs. Given that we group LPs in classes, the `ClassDefinition` concept is linked to the `StructDefinition` concept, allowing to specify the organisation of the simulation state for a specific homogeneous set of LPs in the model. However, specifying the structure of the state is not enough for a DES model: at simulation startup, LPs may need to initialise their state to some initial values. If the `StartupFunction` allows to initialise the model *globally*, LPs could be required to initialise their state *locally*. Different REs may have very different support for this, and in the literature various approaches have been proposed. Some REs may provide custom initialisation events that are scheduled to every LP before the simulation starts, others may require the modeller to explicitly define such events, others may lack such support and require dedicated pre-launch code. We have decided to abstract this complexity and variety of approaches by introducing the aforementioned class `StartupFunction` concept. This concept allows the modeller to concisely specify the logic required to initialise the state of a class of LPs. The M2T transformations will then generate the appropriate machinery to inject this logic into the best-suited location, depending on the RE capabilities and requirements.

Models may also require global variables, which can be regarded as global configuration values that are initialised before the actual simulation start, e.g. to support what-if analysis or exploration of model configuration. This initialisation, which is in charge of the `StartupFunction`, is supported by the `GlobalVariable` concept that allows to declare such variables. Similarly, the `Constant` concept allows to declare constant values, proper of the model's domain, that could be used when processing events.

Finally, literate programming is supported by the `DocsElement` concept. This concept allows to insert, in any point of the DESL program, a `Header` for sectioning the code, some `PlainText`, or some `ItemList`. By using these concepts, documentation can be introduced in any point of a DESL program. Unlike traditional literal programming, where some comments or special markings are required for the tangle and weave processors to behave correctly, using our MODEL-based approach allows a seamless interweave of logic and documentation.

To clarify how a DESL program looks like, we report in Figure 3 the rendering of the DESL program for the traditional PHold benchmark [15].

```
DES Model: Parallel Hold (PHold)

A synthetic benchmark maintaining a constant number of events in the
simulation. Synchronization and communication overhead are isolated,
focusing on the mechanics of event distribution and processing.

Events:

PHold has only a single dummy event.
* EVENT

Constants:

The number of LPs in the model.
#define NUM_LPS = 16000;

Structs:

struct phold_msg {
   int dummy_data;
};

External Functions:

An external busy loop, for avoiding compiler optimizations.
void busy_loop(int max);
The RNG functions that generate event timestamps.
void InitializeRNG(rng_ctx *random_context);
double Exp(double mean);

Configuration:

This is the number of iterations in the busy loop. Mapping this count into a
 timing requires careful benchmarking on the used machine

int loop_count = 150;

Handlers:

Class classA:
 struct phold_state {
  int complete_events;
  rng_ctx rand;
 };

 StartupFunction (int lp_id, phold_state *state) {
  InitializeRNG(&state->rand);
  SendEvent EVENT to lp_id at Exp(mean) with
    <no payload>
 }

 handler EVENT (int lp_id, double now, phold_msg *msg, phold_state *state) {
  busy_loop(loop_count);
  phold_msg new_event = {0};
  SendEvent EVENT to GetRandomNeighbour(lp_id) at now
    + Exp(mean) with new_event
 }

Process Allocation:

assign_class([0, 15999], classA);
```

Figure 3: Implementation of PHold in DESL.

3.2 Additional Concepts

We have included additional concepts in the DESL MODEL, which can be considered syntactic sugar: they are implementable with the base MODEL, but we consider them relevant abstractions for DES modellers. In the following, we detail such additional concepts.

3.2.1 Collections. In DES models, we typically find *collections* of objects. Think, for example, of vehicles and pedestrians in a traffic simulation, or customers and servers in a queueing system, molecules in a chemical reaction model, or tasks and processors in a distributed computing simulation. Models typically have to encapsulate the dynamic entities of the system, which interact through discrete events that govern state transitions and temporal evolution.

Collections are therefore relevant because they are a recurring aspect of DES models. We allow a direct representation of collections in a DESL program including the `Collection` concept. This concept relates to a `TypeDefintion`: collections of generic data structures can be defined and used in event payloads and LP states.

By using a `Collection`, model developers are able to iterate over them using a `foreach` statement, to add new elements to a collection, or to remove an element from a collection by reference. Such operations are typical of object-oriented programming languages, but are not necessarily supported by all REs. This is especially true if the programming language supported by some RE is not object-oriented (e.g., in the case of the C language supported by REs like [9, 19, 31, 36]). In this case, our M2T transformations can take care of supporting `Collections` by injecting in the generated program specific data structures that, otherwise, would require a non-minimal amount of time to be included in the model, without any real justification, and could be error-prone.

Therefore, supporting `Collections` in the DESL MODEL can reduce the time required to implement a model, allowing simulationists to concentrate more on the model's logic.

3.2.2 Connectivity. Another recurring aspect observed in DES models is the *connectivity* between LPs. Think for example of road segments in traffic simulation, or communication links in a network simulation, conveyor belts in a manufacturing process, or data streams between processing nodes in a distributed computing model. Connectivity defines the pathways along which interactions and dependencies propagate, shaping the system's dynamic behaviour, and influencing synchronization, load distribution, and event scheduling.

We have therefore decided to include connectivity capabilities in DESL, restricting its existence only to encapsulation demands, rather than for PDES-inspired reasons. In this way, modellers are able to specify different interconnections between LPs, focusing on the structural organization of their models rather than on performance-driven partitioning strategies. This design choice ensures that connectivity serves as a means of encapsulating logical relationships, such as communication channels, shared resources, or hierarchical dependencies, without imposing constraints dictated by parallel execution concerns. Consequently, modellers can define interaction topologies that reflect the conceptual structure of the system.

The DESL framework supports these connectivity capabilities relying on a dedicated *topology library*. This library allows to define multiple 2D and 3D grids, and also generic graphs with weighted edges. Weights in the edges can also be used to represent the probability that an event is scheduled towards a connected LP. Additionally, every edge can be associated with a model-defined payload, to represent any kind of description of the connection between two LPs. M2T transformations can inject calls to the library whenever needed, and the library itself into the generated model.

In a DESL program, two aspects related to connectivity are captured by the DESL MODEL. First, at simulation startup, some specific topology should be instantiated. This is delegated to the `InitTopology` statement, which can be embedded in a `StartupFunction`. In `InitTopology`, the modeller can pick one specific grid or graph and configure the topology based on the `ProcessAllocation` specified in the DESL program.

A collection can be queried whenever a `SendEvent` concept is used. In this case, the modeller can rely on two different capabilities

```
DES Model: Personal Communication Service (PCS)
The PCS model represents a wireless communication network where mobile
devices move between predefined service areas (cells) and initiate or
receive calls. The network is structured as a grid of cells, each
functioning as an independent processing entity.

Events:
This represents the initialization events, where the simulation state is
defined for every LP.
* LP_INIT

These two events represent the initiation and termination of a mobile call.
* START_CALL
* END_CALL

When the user crosses the boundary of a base station and the call is still
ongoing, we transfer the call from the source toward the destination base
station. These two events split the transfer of the call between the source
(LEAVE) and destination (RECV) stations.
* HANDOFF_RECV
* HANDOFF_LEAVE

The model performs accurate simulation of the signal quality, also
considering a meteorological model. Periodically, the signal strength and
power are recomputed accounting for whether changes. This event is used for
triggering the power consumption computation.
* FADING_RECHECK

Constants:
Maximum number of wireless channels in a cell.
#define CHANNELS_PER_CELL = 1000;

New calls are initiated following a Poisson process. This represents the
mean interarrival time separating consecutive calls.
#define TA = 0.12;

Average duration of a call. Actual durations are drawn from an exponential
distribution.
#define TA_DURATION = 60;
```

Figure 4: Excerpt of the source code of the PCS model.

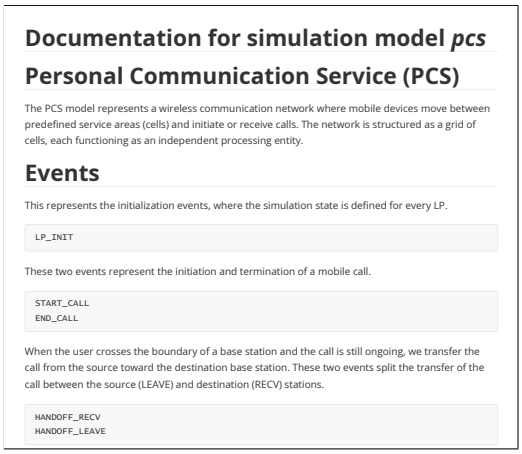
of our connectivity abstraction. A *random* receiver can be selected, from the pool of *adjacent* LPs. In case of a 2D/3D grid, the adjacency depends on the specific type of grid. In case of a graph, randomness accounts also for the weight/probabilities associated with edges. If randomness is not required by the model, it is possible to retrieve a `Collection` of all possible neighbours of an LP, and implement some custom scheduling logic, e.g. based on model-defined payload associated with graph edges.

As mentioned, DESL is not a DSL, so we do not consider any specific domain in the connectivity abstraction. Connectivity should support a large number of different domains. This can be beneficial because, as we mentioned before, it is possible to use DESL also as the target of a M2M transformation: if some specific domain requires some specific topology, it is possible to rely on the `Connectivity` concept of DESL to easily generate working models, with reduced hassle for DSL designers.

3.3 Model to Text Transformations

To showcase the viability of the DESL framework, and in particular to highlight the flexibility of its model, we have implemented some M2T transformations that allow executing DELS programs on some REs from the literature, and to generate the documentation of the model. The REs that we have supported in this work are ROSS [9], ROOT-Sim [36], and USE [19]. In this section, we discuss the M2T transformations to generate working models for these REs.

3.3.1 The Weave Transformation. The classic approach to the weave processor is to transform the initial source code in some different sources that could be compiled to generate the final documentation. We retain this approach: our M2T weave transformation transforms



Documentation for simulation model *pcs*

Personal Communication Service (PCS)

The PCS model represents a wireless communication network where mobile devices move between predefined service areas (cells) and initiate or receive calls. The network is structured as a grid of cells, each functioning as an independent processing entity.

Events

This represents the initialization events, where the simulation state is defined for every LP.

```
LP_INIT
```

These two events represent the initiation and termination of a mobile call.

```
START_CALL
END_CALL
```

When the user crosses the boundary of a base station and the call is still ongoing, we transfer the call from the source toward the destination base station. These two events split the transfer of the call between the source (LEAVE) and destination (RECV) stations.

```
HANDOFF_RECV
HANDOFF_LEAVE
```

Figure 5: Excerpt of the generated documentation.

a DESL program into a markdown file that could be later converted to other formats using existing tools. The weave M2T transformation is quite simple, and is based on the following specification:

```
text gen component for concept DocumentationM2T {
file name : (node)→string {
  node.name;
}
file path : <model/qualified/name>
extension : (node)→string { "md"; }
  (node)→void {
    append {# Documentation for *} ${node.name} {*} \n;
    append create docs component node.events;
    append create docs component node.constants;
    append create docs component node.typedefs;
    append create docs component node.structs;
    append create docs component node.externalFunctions;
    append create docs component node.configuration;
    append create docs component node.startup;
    append create docs component node.classes;
    append create docs component node.processAllocations;
  }
}
```

Essentially, this M2T transformation iterates over each concept defined in the DESL model and, for each concept defined in a DESL program, it invokes the following transformation:

```
text gen component for concept DocsEntry {
  (node)→void {
    foreach element in node.elements {
      append ${element} \n ;
    }
  }
}
```

which emits each concept as plain text in the output file. During this part of the generation, each concept is either included as a code block (in the case of model logic) or as plain text or a section, depending on the specific type of `DocsElement` concept that is being processed. In this way, the distinction between documentation concepts and code concepts are transformed into different markups, ensuring that model logic is properly encapsulated within a structured representation while textual documentation retains its readability and hierarchical organization.

To illustrate the output of the weave M2T transformation, we present in Figure 4 an excerpt of the DESL program of the Personal Communication System (PCS) [24], a model of mobile calls. An excerpt of the generated documentation is shown in Figure 5. As can be seen, the transformation process effectively differentiates

between model logic and explanatory content, ensuring that the generated documentation maintains both structural coherence and semantic clarity. Conceptual descriptions, section headers, and inline explanations are seamlessly integrated with the corresponding code segments. This approach ensures that both domain experts and developers can navigate, interpret, and refine the model efficiently.

3.3.2 The ROSS Tangle Transformation. To generate working code for ROSS, there are two main aspects that we have to deal with in the M2T transformation. First, ROSS works as a simulation library, requiring the model's code to setup the library before activating the actual simulation. This organization of the model well resembles the organization of the DESL MODEL, where the initialization logic and the actual model's logic are represented with different concepts—the global `StartupFunction` and the `EventHandlers`. Therefore, the main part of the ROSS M2T transformation (which we do not report in the paper for space constraints) defines a `main` function where all the machinery to configure and activate ROSS is included. The `StartupFunction` is injected in the `main` before the actual simulation is started.

LPs in ROSS are defined by relying on several callback functions. The M2T transformation populates such callback functions based on their rationale. For each LP, we register a forward event handler, that is not the materialization of an `EventHandler`. Indeed, this handler works as an event dispatcher to implement the `ClassDefinition` concept. In fact, according to the DESL MODEL, an LP belongs to a class, based on the `ProcessAllocation` concept. The ROSS handler, therefore, determines the class to which the LP belongs (based on its id), and only then determines (based on the event type) what is the actual `EventHandler` to be activated. In this way, different handlers can be activated according to the model's configuration, possibly determined by the global `StartupFunction`.

We rely on the same approach for the classes `StartupFunctions`. ROSS provides initialization callbacks, that are executed before the actual simulation starts. These callbacks are used in the exact same way: they serve as class dispatchers, to activate the corresponding `StartupFunction`.

ROSS employs a convenient and versatile way to allocate LPs to processing elements (PEs), which are then mapped to MPI ranks. In our M2T transformation, we generate a custom allocation function that maps LPs to PEs in blocks. While other strategies could be coded by the modeller, which could show different performance profiles depending on the communication patterns, we have used this kind of allocation mainly because it is similar to what the other REs do in their initialization, thus allowing us to carry out a more fair performance comparison.

A relevant capability of ROSS is its support of rollbacks by means of reverse computation. An LP can register a reverse handler callback that is activated upon a rollback operation. This callback receives the event that was processed in forward execution, and can use it to execute the reverse logic to undo its updates on the LP simulation state. Currently, we are working on M2M transformations to automatically generate reverse event logic, but this approach is complex and is out of the scope of this paper. For this paper, we have actually exploited one capability of reverse computation of ROSS to implement checkpoint-based rollbacks.

In fact, ROSS is aware that not all operations are reversible—*destructive* operations such as assignments cannot be undone. For this reason, ROSS has also minimalistic checkpointing capabilities: an event can be modified in forward execution, storing any kind of data in it. These data are then observable in the reverse handler, upon a rollback. This approach was originally used in [9] to store the old value of assignments, or to keep track of what branches in complex control flows where traversed in forward execution. We use this capability to store the old values of the state in forward execution, and then reinstall them upon a rollback.

ROSS also offers support for conservative synchronization. In this case, it is necessary to specify the lookahead of the model at startup. This operation is demanded from the `StartupFunction`, in which the modeller can do so by relying on an `ExternalFunction` concept. Abstracting away the concept of lookahead would be straightforward in DESL, but we have not yet identified a strategy common to other REs—ROSS is the only considered RE with such functionality—and will require further investigation to determine whether a generalised approach can be integrated into DESL without introducing unnecessary complexity.

3.3.3 The ROOT-Sim Tangle Transformation. The organization of a model in ROOT-Sim is similar to ROSS, except for the lack of reverse computation support. Indeed, also ROOT-Sim can be used as a simulation library, where the model's code is in charge of setting up the model and configuring the library before activating the simulation. In this sense, in the M2T transformation for ROOT-Sim, we generate a `main` program that is essentially the equivalent of what we have in ROSS.

The main difference is that LPs are not defined by dedicated callback functions. Conversely, ROOT-Sim has a single generic callback function, named `ProcessEvent`, that is activated for every LP and every event type. In this sense, this callback function should be regarded as a generic event dispatcher, and we abide by this organization in the M2T transformation.

For LP class initialization, there is one special event, called `LP_INIT`, that is scheduled at every LP before the actual simulation starts. This is the point in which we can inject the per-class `StartupFunction` of a DESL program. Therefore, we inject in the `ProcessEvent` dispatcher the code that, depending on the LP's id, determines the class of the LP (based on the `ProcessAllocation` concept). After the class of an LP is determined, we either activate the `StartupFunction` in case of an `LP_INIT` event, or the proper `EventHandler` in case a different event is scheduled by the RE.

ROOT-Sim handles rollbacks transparently. The only caveat is to rely on the internal memory allocations functions. Doing so is trivial, as every time that an object is allocated in DESL, we simply divert the allocation to the internal ROOT-Sim's memory allocator.

3.3.4 The USE Tangle Transformation. USE has a different approach than ROSS and ROOT-Sim at simulation configuration and rollback management. In particular, USE considers the model as a form of *plugin*, in the sense that the entire application startup is handled by its internal `main` program. The simulation is therefore started before the actual model's code takes control, which makes the model's configuration from a DESL program more difficult.

At the same time, USE offers an LP initialization capability similar to ROOT-Sim's, where a dedicate initialization event is scheduled

at every LP, in LP id order, before the actual simulation starts. We have therefore decided to inject the global `StartupFunction` in this event, only when the first LP of the first class is activated. In this way, model configuration is carried out before the actual simulation starts, and before the class `StartupFunctions` are executed.

The rest of the organization of USE matches that of ROOT-Sim, so we have applied the same strategy in the M2T transformation.

Regarding optimistic synchronization support, memory management is USE is also transparent, but the model's code is instrumented at compile time, redirecting memory allocation to the internal memory allocator. We have exploited this functionality by using standard `malloc` calls in the M2T transformation, letting the USE compiling toolchain instrument the generated model code to support memory recoverability.

4 Experimental Assessment

In this Section, we provide some preliminary performance results collected using the code generated using the *tangle* transformations discussed above. We emphasise that the actual figures obtained *are not extremely important* (see Section 5): the most notable result is actually the ability to have collected these numbers without any manual intervention on the model code. At the same time, from the result some of the claims of this paper become evident.

4.1 Execution Environment and Benchmarks

We have executed our experiments on a machine equipped with an Intel(R) Xeon(R) Silver 4210R CPU, with 20 physical cores (40 hyperthreads) running at 2.40GHz, and 160 GB of RAM. The machine has two NUMA nodes. We have used the already mentioned DES models, namely PCS [24] and PHold [15].

PCS models a GSM-based mobile network through a detailed simulation where each LP simulates the evolution of an individual hexagonal cell, collectively covering a square region. The simulation is high fidelity, managing a parameterizable number of wireless channels per cell, incorporating explicit models for power regulation and the complex phenomena of interference and fading, according to the specification in [24]. When a call is initiated, a dynamically allocated call-setup record is linked to the cell's active call list, which is later removed when the call terminates or when a handoff to an adjacent cell occurs, triggering an analogous setup at the destination. Power regulation during call setup involves scanning the active records to compute the minimum transmission power required to meet a predefined signal-to-interference ratio, while data structures that track fading coefficients are updated in response to a meteorologically based model of climatic conditions.

The behaviour of the model is influenced by parameters such as the inter-arrival time of calls, the expected call duration, and the residual time a device remains in a cell, which together determine a utilization factor affecting computational load and memory requirements. Higher channel utilization results in more frequent allocation (i.e., larger state memory footprint) and scanning of call management records (i.e., larger-grain events). In our simulations, we have considered three utilization factors, associated with 25% of channel occupation (*light* load), 50% (*medium* load), and 75% (*heavy* load). We have run simulations with 22,500 LPs, simulating a total of 10 minutes (logical time) at steady state.

The second benchmark we have used in our experiments is PHold [15], a traditional synthetic benchmark from the literature, aimed at evaluating PDES REs. PHold generates a continuous stream of simulation events across the LPs: every executed event triggers the injection of another event, scheduled to a random LP with a random timestamp in the future. Moreover, PHold is parameterizable in event duration: a busy loop is included in the execution of the event, allowing to control the latency of each event, in a domain-independent fashion. In this way, PHold can be used to measure the scalability and efficiency of PDES RE implementations. We have run our experiments using 16,000 LPs, using three different busy loop durations for each event: $1\mu s$, $10\mu s$, and $100\mu s$.

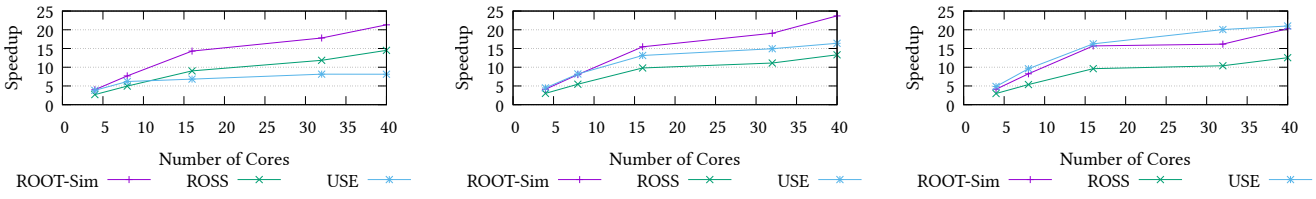
4.2 Results

All results presented in this study are averaged over 5 runs. In Figure 6 we report the speedup for the PCS benchmark over an optimized sequential execution based on a calendar queue [6]. All the considered REs scale when the number of cores is increased, thus showing their capability to harness parallelism from the model. Interestingly, ROSS and ROOT-Sim are resilient to the event duration, with scalability curves that are mostly the same for the three different loads we have considered, with ROOT-Sim having a higher performance than ROSS. Conversely, USE shows a limited scalability in case of fine-grained events, while for larger event durations it is able to compete with the two other considered REs.

The reason for this result lies in the inner organization of the REs. ROSS and ROOT-Sim have a stronger adherence to the original conception of a Time Warp synchronization scheme [21]. Conversely, USE relies on heavyweight event management data structures, whose goal is to prefer the execution of safe events to events that have a higher rollback probability [28]. Moreover, USE has a very infrequent *fossil collection* execution, because safe events are immediately discarded, à la conservative synchronization. Longer events show different dynamics—PCS events have been observed to have a granularity of up to $100\mu s$. In this case, even a small rollback probability—in our experiments ROOT-Sim has an efficiency of 98% in the worst-case scenario—can have non-negligible secondary effects on the caching hierarchy. Also, the heavier fossil collection phase imposes additional memory management overhead by triggering large-scale cache invalidations and synchronization delays that, when combined with even a minimal rollback probability, further degrades overall simulation throughput and scalability.

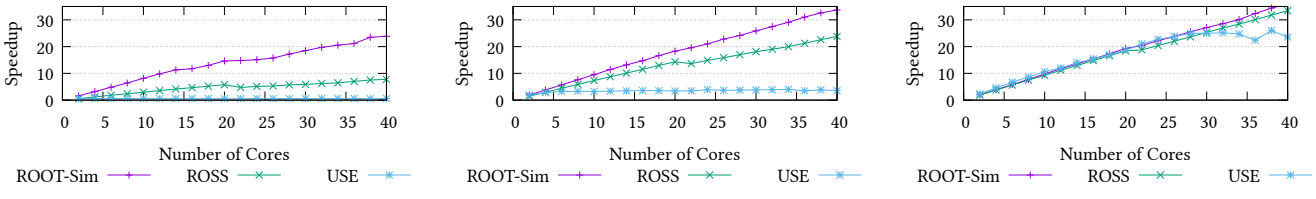
These results are confirmed by the PHold model benchmark, as reported in Figure 7. From the results, we observe that for finer-grain events, USE is not able to provide any significant speedup—often, the speedup is < 1. Conversely, as soon as the event duration increases, USE's performance improves, until it is able to outperform ROSS and ROOT-Sim, although slightly. Both ROSS and ROOT-Sim, conversely, are able to scale mostly linearly, with minor fluctuations observed once the second NUMA node begins to be used. This is an expected result, as PHold has been extensively used in the literature to benchmark these simulators.

Overall, these results confirm what has already been observed in the literature: there is no single RE or algorithm that can benefit all models or all models' configurations. Our MDE-based approach allows to switch from one RE to another seamlessly, without any



(a) *Light* load. **(b) *Medium* load.** **(c) *Heavy* load.**

Figure 6: Comparative Speedup for the PCS benchmark



(a) 1 μs busy loop. **(b) 10 μs busy loop.** **(c) 100 μs busy loop.**

Figure 7: Comparative Speedup for the PHold benchmark

need for manual intervention. This capability has a significant advantage in the case of extensive simulation studies, where a large number of model's parameters are used, and switching from one RE to another could improve the overall performance of the study.

5 Threats to Validity

The M2T transformations employed in this work may potentially threaten the validity of the presented experimental results. One such threat is that the transformations may not produce the most efficient code for a given RE. In particular, an expert familiar with a specific RE might be able to manually write more efficient code than that produced by the current transformations. This discrepancy could affect the relative speedup trends presented in Section 4. To address this, we plan to conduct a more thorough performance optimization of the transformations, aiming to ensure that the generated code is as efficient as possible. Additionally, we hope that maintainers of the currently supported REs, as well as those supported in the future, will collaborate with the project to further enhance the performance of the generated code.

Another threat comes from the lack of validation of M2T transformations. There remains a risk that some transformations may generate incorrect code. To mitigate this, we have manually inspected the correctness of the generated source code and validated the results of simulation executions, as a simulation validity argument [51]. However, a formal evaluation of the transformations is necessary to enhance the trustworthiness of the generated code. Such an evaluation would provide greater confidence in the reliability and correctness of the system.

6 Conclusions and Future Work

In this work, we have presented DESL, an MDE-based approach to generate from the same source code base, conforming to a MODEL, the source code to be executed on different PDES REs, and the associated documentation. We have shown that it is possible to rely on ad-hoc M2T transformations to circumvent the specific characteristics and expectations of the REs on the specific model construction approach and produce working code. The resulting code also has good performance and scalability, highlighting that MDE is a viable solution to hide away the complexity of model development while providing competitive simulation executions.

Furthermore, the experimental assessment has highlighted, as already stressed in recent literature, that there is not a single algorithm or RE that can fit all models' requirements. Therefore, the interoperability capabilities of DESL and MDE in general are a viable solution supporting model developers to focus on implementing their models, while avoiding any lock in to a specific technology or RE. This approach can sustain higher performance simulation studies allowing, e.g., to switch to different algorithms or REs also based on the model's configuration parameters.

Future work will span across various directions. We plan to support additional REs, experiment with additional real-world models, and introduce intermediate M2M transformations that could support the formal verification of the correctness and equivalence of the generated sources.

Acknowledgments

This paper has been partially supported by European Union—Next Generation EU, Mission 4, Component 2, CUP E53D23008200006, and partially by the Spoke 1 "FutureHPC & BigData" funded by European Union—Next Generation EU, Mission 4, Component 2.

We also thank anonymous reviewer #2 for their constructive and detailed feedback, a rarity in current peer-review practices.

References

[1] Philipp Andelfinger, Till Köster, and Adelinde M Uhrmacher. 2023. Zero lookahead? Zero problem. The window racer algorithm. In *ACM SIGSIM Conference*

on Principles of Advanced Discrete Simulation (PADS '23). ACM, New York, NY, USA, 1–11. doi:10.1145/3573900.3591115

[2] Philipp Andelfinger and Adelinde M Uhrmacher. 2023. Synchronous speculative simulation of tightly coupled agents in continuous time on CPUs and GPUs. *International Conference on Advances in System Simulation* 100 (March 2023), 5–21. doi:10.1177/00375497231158930

[3] María Julia Blas, Silvio Gonnet, Doohwan Kim, and Bernard P Zeigler. 2023. A context-free grammar for generating full Classic DEVS models. In *Proceedings of the 2023 Winter Simulation Conference (WSC '23)*. IEEE Press, Piscataway, NJ, USA, 2579–2590. doi:10.1109/WSC60868.2023.10407991

[4] Vincent A M Bonnet. 2017. *Benchmarking Parallel Discrete Event Simulations*. Ph. D. Dissertation. Utrecht University.

[5] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-driven software engineering in practice*. Springer International Publishing, Cham. doi:10.1007/978-3-031-02549-5

[6] Randy Brown. 1988. Calendar Queues: a Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Commun. ACM* 31 (1988), 1220–1227.

[7] David Bruce. 1997. What makes a good domain-specific language? Apostle, and its approach to parallel discrete event simulation. In *Proceedings of the 1st ACM SIGPLAN Workshop on Domain-Specific Languages (DSL'97)*. ACM, New York, NY, USA, 19.

[8] Wentong Cai, Christopher Carothers, David M Nicol, and Adelinde M Uhrmacher. 2023. *Computer Science Methods for effective and Sustainable Simulation Studies (Dagstuhl Seminar 22401)*. Technical Report. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany. 1–60 pages. doi:10.4230/DAGREP.12.10.1

[9] Christopher Carothers, David Bauer, and Shawn Pearce. 2002. ROSS: A high-performance, low-memory, modular Time Warp system. *Journal of parallel and distributed computing* 62, 11 (Nov. 2002), 1648–1669. doi:10.1016/S0743-7315(02)00004-7

[10] K Mani Chandy and Jayadev Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM* 24, 4 (April 1981), 198–206. doi:10.1145/358598.358613

[11] Li-Li Chen, Ya-Shuai Lu, Yi-Ping Yao, Shao-Liang Peng, and Ling-da Wu. 2011. A Well-Balanced Time Warp System on Multi-Core Environments. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*. IEEE, Piscataway, NJ, USA, 1–9. doi:10.1109/PADS.2011.5936752

[12] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2017. Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES. *ACM transactions on modeling and computer simulation: a publication of the Association for Computing Machinery* 27, 2 (April 2017), 1–26. doi:10.1145/3077583

[13] Biagio Cosenza, Nikita Popov, Ben Juurlink, Paul Richmond, Mozhgan Kabiri Chimeh, Carmine Spagnuolo, Gennaro Cordasco, and Vittorio Scarano. 2018. OpenABL: A domain-specific language for parallel and distributed agent-based simulations. In *Euro-Par 2018: Parallel Processing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer International Publishing, Cham, 505–518. doi:10.1007/978-3-319-96983-1_36

[14] Richard M Fujimoto. 1990. Parallel Discrete Event Simulation. *Commun. ACM* 33, 10 (Oct. 1990), 30–53. doi:10.1145/84537.84545

[15] Richard M Fujimoto. 1990. Performance of Time Warp Under Synthetic Workloads. In *Distributed Simulation (PADS'90)*, David Nicol (Ed.). Society for Computer Simulation International, San Diego, CA, USA, 23–28.

[16] Marc-Oliver Gewaltig and Markus Diesmann. 2007. *NEST (NEural Simulation Tool)*. Vol. 2. Scholarpedia, Chapter 4. doi:10.4249/scholarpedia.1430

[17] Volker Grimm, Jacqueline Augusiak, Andreas Focks, Béatrice M Frank, Faten Gabsi, Alice S A Johnston, Chun Liu, Benjamin T Martin, Mattia Meli, Viktoriia Radchuk, Pernille Thorbek, and Steven F Railsback. 2014. Towards better modelling and decision support: Documenting model development, testing, and analysis using TRACE. *Ecological modelling* 280 (May 2014), 129–139. doi:10.1016/j.ecolmodel.2014.01.018

[18] Jan Himmelspach and Adelinde M Uhrmacher. 2007. Plug'n Simulate. In *Proceedings of the 40th Annual Simulation Symposium*. IEEE, Piscataway, NJ, USA, 137–143. doi:10.1109/anss.2007.34

[19] Mauro Ianni, Romolo Marotta, Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2018. The Ultimate Share-Everything PDES System. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '18)*. ACM, New York, NY, USA, 73–84. doi:10.1145/3200921.3200931

[20] IEEE Standards Association. 2010. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules.

[21] David R Jefferson. 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 404–425. doi:10.1145/3916.3988

[22] David R Jefferson. 1990. Virtual time II: Storage Management in Conservative and Optimistic Systems. In *Proceedings of the 9th Symposium on Principles of Distributed Computing (PODC '90)*. ACM, New York, NY, USA, 75–89. doi:10.1145/93385.93403

[23] David R Jefferson and Peter D Barnes, Jr. 2022. Virtual time III, Part 1: Unified Virtual Time synchronization for parallel discrete event simulation. *ACM transactions on modeling and computer simulation: a publication of the Association for Computing Machinery* 32, 4 (Oct. 2022), 1–29. doi:10.1145/3505248

[24] Sunil Kandukuri and Stephen Boyd. 2002. Optimal Power Control in Interference-Limited Fading Wireless Channels with Outage-Probability Specifications. *IEEE Transactions on Wireless Communications* 1 (2002), 46–55.

[25] Donald E Knuth. 1984. Literate Programming. *The computer journal* 27, 2 (Feb. 1984), 97–111. doi:10.1093/comjnl/27.2.97

[26] Till Köster, Adelinde M Uhrmacher, and Philipp Andelfinger. 2022. Towards an open repository for reproducible performance comparison of parallel and distributed discrete-event simulators. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '22)*. ACM, New York, NY, USA, 31–32. doi:10.1145/3518997.3534989

[27] Till Köster, Tom Warnke, and Adelinde M Uhrmacher. 2020. Partial evaluation via code generation for static stochastic reaction network models. In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, New York, NY, USA, 159–170. doi:10.1145/3384441.3395983

[28] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2024. A conflict-resilient lock-free linearizable calendar queue. *ACM transactions on parallel computing* 11, 1 (March 2024), 1–32. doi:10.1145/3635163

[29] Romolo Marotta and Alessandro Pellegrini. 2024. Model-Driven Engineering for High-Performance Parallel Discrete Event Simulations on Heterogeneous Architectures. In *Proceedings of the 2024 Winter Simulation Conference (WSC '24)*, H Lam, E Azar, D Batur, S Gao, W Xie, S R Hunter, and M D Rossetti (Eds.). IEEE, Piscataway, NJ, USA, 2202–2213. doi:10.1109/WSC63780.2024.10838978

[30] Romolo Marotta, Alessandro Pellegrini, and Philipp Andelfinger. 2024. Follow the leader: Alternating CPU/GPU computations in PDES. In *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS'24)*. ACM, New York, NY, USA, 47–51. doi:10.1145/3615979.3656056

[31] Dale E Martin, Timothy J McBrayer, and Philip A Wilsey. 1996. WARPED: a time warp simulation kernel for analysis and application development. In *Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS, Vol. 1)*. IEEE Computer Society, Piscataway, NJ, USA, 383–386 vol.1. doi:10.1109/HICSS.1996.495485

[32] Tom Mens and Pieter Van Gorp. 2006. A taxonomy of model transformation. *Electronic notes in theoretical computer science* 152 (March 2006), 125–142. doi:10.1016/j.entcs.2005.10.021

[33] Quang Anh Pham Nguyen, Philipp Andelfinger, Wentong Cai, and Alois 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)*. ACM, New York, NY, USA, 115–126. doi:10.1145/3316480.3322893

[34] Eunjung Park, Stephan Eidenbenz, Nandakishore Santhi, Guillaume Chapuis, and Bradley Settlemyer. 2015. Parameterized benchmarking of parallel discrete event simulation systems: Communication, computation, and memory. In *2015 Winter Simulation Conference (WSC)*. IEEE, Piscataway, NJ, USA, 2836–2847. doi:10.1109/WSC.2015.7408388

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

[36] Alessandro Pellegrini, Roberto Vitali, and Francesco 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)*. ICST, Brussels, Belgium, 96–98. doi:10.4108/icst.simutools.2011.245551

[37] Andrea Piccione, Philipp Andelfinger, and Alessandro Pellegrini. 2023. Hybrid Speculative Synchronisation for Parallel Discrete Event Simulation. In *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '23)*. Association for Computing Machinery, New York, NY, USA, 84–95. doi:10.1145/3573900.3591124

[38] Markus Schordan, Tomas Oppelstrup, David R Jefferson, Peter D Barnes, and Daniel Quinlan. 2016. Automatic Generation of Reversible C++ Code and Its Performance in a Scalable Kinetic Monte-Carlo Application. In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '16)*. ACM, New York, NY, USA, 111–122. doi:10.1145/2901378.2901394

[39] David Shuttleworth and Jose J Padilla. 2021. Towards semi-automatic model specification. In *2021 Winter Simulation Conference (WSC)*. IEEE, Piscataway, NJ, USA, 1–12. doi:10.1109/wsc52266.2021.9715393

[40] Jeffrey S Steinman. 1991. SPEEDES: A Unified Approach to Parallel Simulation. In *Advances in Parallel and Distributed Simulation (PADS '91)*, Vijay K Madisetti, David Nicol, and Richard M Fujimoto (Eds.). Society for Computer Simulation, San Diego, CA, USA, 1111–1115.

[41] Jeffrey S Steinman. 1993. Breathing Time Warp. *Simuletter* 23, 1 (July 1993), 109–118. doi:10.1145/174134.158473

[42] Brian Paul Swenson and George F Riley. 2012. A new approach to Zero-Copy message passing with reversible memory allocation in multi-core architectures. In *Proceedings of the ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation (PADS'12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 44–52. doi:10.1109/PADS.2012.3

[43] Adelinde M Uhrmacher, Peter Frazier, Reiner Hähnle, Franziska Klügl, Fabian Lorig, Bertram Ludäscher, Laura Nenzi, Cristina Ruiz-Martin, Bernhard Rumpe, Claudia Szabo, Gabriel Wainer, and Pia Wilsdorf. 2024. Context, composition, automation, and communication: The C ^{2} AC roadmap for modeling and simulation. *ACM transactions on modeling and computer simulation: a publication of the Association for Computing Machinery* 34, 4 (Oct. 2024), 1–51. doi:10.1145/3673226

[44] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2009. Benchmarking Memory Management Capabilities within ROOT-Sim. In *Proceedings of the 13th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, Piscataway, NJ, USA, 33–40. doi:10.1109/DS-RT.2009.15

[45] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2012. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 26th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS'12)*. IEEE, Piscataway, NJ, USA, 211–220. doi:10.1109/pads.2012.46

[46] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. 2012. 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 (SPLASH '12)*. ACM, New York, NY, USA, 121–140. doi:10.1145/2384716.2384767

[47] Jingjing Wang, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2015. AIR: Application-level interference resilience for PDES on multicore systems. *ACM transactions on modeling and computer simulation: a publication of the Association for Computing Machinery* 25, 3 (May 2015), 1–25. doi:10.1145/2701420

[48] Jingjing Wang, Deepak Jagtap, Nael B Abu-Ghazaleh, and Dmitry Ponomarev. 2014. Parallel discrete event simulation for multi-core systems: Analysis and optimization. *IEEE Transactions on Parallel and Distributed Systems* 25 (2014), 1574–1584. doi:10.1109/TPDS.2013.193

[49] Pia Wilsdorf, Jakob Heller, Kai Budde, Julius Zimmermann, Tom Warnke, Christian Haubelt, Dirk Timmermann, Ursula van Rienen, and Adelinde M Uhrmacher. 2022. A model-driven approach for conducting simulation experiments. *Applied sciences (Basel, Switzerland)* 12, 16 (Aug. 2022), 7977. doi:10.3390/app12167977

[50] Pia Wilsdorf, Anja Wolpers, Jason Hilton, Fiete Haack, and Adelinde M Uhrmacher. 2022. Automatic Reuse, Adaption, and Execution of Simulation Experiments via Provenance Patterns. *ACM Transactions on Modeling and Computer Simulation* 33, 1-2 (Sept. 2022), 1–27. doi:10.1145/3564928

[51] Pia Wilsdorf, Steffen Zschaler, Fiete Haack, and Adelinde M Uhrmacher. 2024. Potential and Challenges of Assurance Cases for Simulation Validation. In *Proceedings of the 2024 Winter Simulation Conference (WSC '24)*. IEEE, Piscataway, NJ, USA, 2166–2177. doi:10.1109/wsc63780.2024.10838818

[52] Bernard P Zeigler, Tag Gon Kim, and Herbert Praehofer. 2000. *Theory of Modeling and Simulation*. Academic Press, London, UK. doi:10.1016/C2016-0-03987-6

[53] Steffen Zschaler and Fiona A C Polack. 2023. Trustworthy agent-based simulation: the case for domain-specific modelling languages. *Software & Systems Modeling* 22, 2 (Feb. 2023), 455–470. doi:10.1007/s10270-023-01082-9