Model-Driven Engineering for Reversible Parallel Discrete Event Simulation

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

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

***Abstract*—This paper introduces a Model-Driven Engineering approach to automatically generate reverse event handlers for optimistic Parallel Discrete Event Simulation. Optimistic synchronisation, such as Time Warp, relies on the ability to roll back the simulation state to a previous consistent point. Traditionally, this is achieved through manual implementation of reverse handlers or full state checkpointing, both of which are error-prone or computationally expensive.**

The proposed methodology uses model-to-model transformations to derive a reverse model from a high-level forward event model defined in the DESL language. This transformation transparently handles data and structural irreversibility by automatically inverting reversible operations and injecting checkpointing logic for destructive operations. We also address the limitations of existing reverse computing solutions regarding dynamic memory by introducing a reference implementation of a reversible memory allocator specifically designed for PDES. The approach is validated using the ROSS runtime environment, demonstrating that higher-level abstractions can simplify model development while maintaining execution consistency and performance.

***Index Terms*—Model Driven Engineering; Reversible Computing; Parallel Discrete Event Simulation; Time Warp**

I. Introduction

Parallel Discrete Event Simulation (PDES) is a method for running a single Discrete Event Simulation (DES) model on a parallel computer or cluster [1]. The interest in PDES stems from its ability to reduce the enormous time required by DES models to deliver solutions. A fundamental requirement for PDES systems is that, for a given model, their results must be identical to those of a sequential DES model [2], [3]. The inherent nondeterminism of parallel computing requires careful synchronisation among the compute elements that run the simulation. In PDES, two main families of synchronisation algorithms have been devised: *conservative* (e.g., [4], [5]) and *optimistic* (e.g., [6]–[9]).

Optimistic synchronisation is interesting because it can *harness at runtime* the substantial parallelism inherent in many simulation models. Different optimistic synchronisation algorithms employ different degrees of *speculation* [10], where highly aggressive synchronisation algorithms may (temporarily) bring the simulation into an inconsistent state. In this case, to correct the simulation trajectory, the source of the inconsistency must be removed from the execution. Generally speaking, a model must be *reversible* for execution on a speculative PDES system, so that it can undo the effects on the simulation state of executing incorrect events.

Reversible computing (RC) [11]–[13] requires state transition functions to be *bijective*, meaning they possess a unique inverse. However, standard computing systems rely on non-bijective, *destructive* operations (e.g., variable assignments or resource deallocations) that destroy information. According to Landauer's principle [14], this creates an "entropy gap" preventing natural reversal.

Traditionally, optimistic PDES bridges this gap via checkpoint/restore-based *rollback operations* [6], which save full snapshots of the Logical Process (LP) state. Since event handlers often update only tiny portions of large simulation states, full snapshots can be inefficient. To reduce this cost, reverse computation for Time Warp was proposed [15]. This approach reconstructs previous states by reversing non-destructive operations and relying on targeted checkpointing only for destructive ones, yielding better performance.

However, manually creating inverse event handlers e^{-1} from e burdens developers and is highly error-prone [16]. Existing automated generation approaches (e.g., [17], [18]) rely on complex compilation or source-to-source transformations and have limitations. For instance, early proposals [15] prohibited dynamic memory allocation. This is because reversing allocations requires reconstructing the exact previous heap layout to maintain consistency of linked data structures. While forbidding dynamic memory avoids this issue, it severely limits development freedom. Similarly, source-to-source transformation approaches (see, e.g., [39], [53]) operate at the Abstract Syntax Tree (AST) level using compiler infrastructures such as ROSE [40]. While later variants employ compile-time static analysis to exclude local variables from instrumentation [53], these approaches still operate within the semantic scope of the target language. This implies that the logging of address-value pairs can scale poorly with the complexity of features like pointer indirection or templates, and that runtime checks may be needed to manage stack/heap persistence, as the compiler lacks the structural knowledge to isolate semantically critical state changes from trivial memory operations.

A recent research trend has highlighted the viability of relying on Model-driven Engineering [19] (MDE) techniques to support different aspects of simulation model development or execution, also in the context of PDES (see, e.g., [20]–

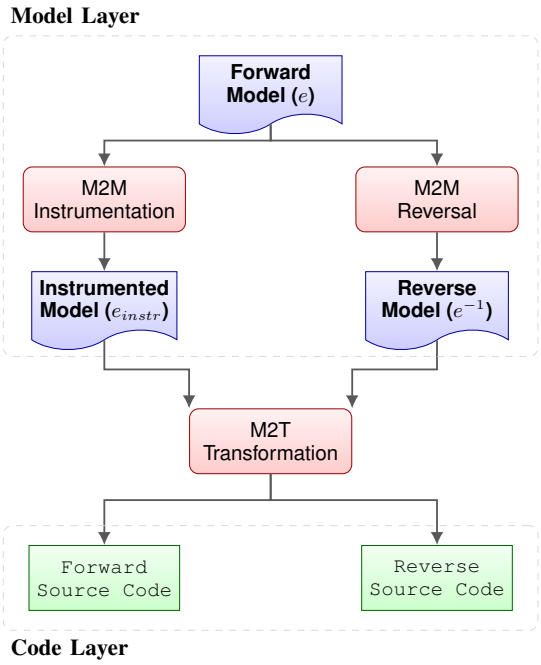

Fig. 1: Automatic generation of reverse event handlers: the forward handler e is fed into a dedicated model-to-model transformation that produces a new model e^{-1} that, if executed, reverts all the changes made on the simulation state by e.

[23]). In MDE, the focus is on *models*, which allow one to specify how the desired system works (a simulation model in our context), independently of any specific programming language, runtime environment or hardware platform. These models can then be transformed into other models via the so-called *model-to-model* (M2M) transformations, or translated into working code via *model-to-text* (M2T) transformations.

M2M transformations can automatically generate a reverse-event model from a forward model. However, since the generated code typically runs on non-reversible systems, creating a reverse handler e^{-1} from a forward model e requires either: (1) *restricting the forward model* [24] to reversible primitives; (2) using *constraint-based inversion* [25], which is non-viable for PDES as it might yield multiple pre-states rather than one unique previous consistent state; or (3) *augmenting the forward model* [26] to instrument the code and log lost information, making the function injective.

In this work, we adopt solution (3). Our contributions are as follows. (i) We introduce two M2M transformations for the DESL language [23] (Figure 1), targeting the ROSS runtime environment [27]. Our transformations automatically generate an instrumented forward model e_{instr} and a reverse model e^{-1} from a single event specification e. The forward model transparently checkpoints variables updated by destructive operations into ROSS events, while the reverse model uses these checkpoints to restore state and automatically inverts non-destructive operations. (ii) We provide a reference implementation of a reversible malloc suited for PDES systems. Unlike other malloc implementations for PDES [28], [29], we do not checkpoint all the state to be restored: our reversible malloc deterministically reconstructs the *structure* of the dynamic memory, while the state *content* is reconstructed by the reverse handlers. (iii) We validate our approach on synthetic (PHold) and real-world (PCS) benchmarks, verifying simulation correctness through the COMPADS model [54]. (iv) We release our M2M transformation as open source software[1]. The remainder of this paper is structured as follows. In Section II we discuss related work. The MDE-driven approach to generating reverse event handlers is presented in Section III. An experimental evaluation of our proposal, using synthetic and real-world models, is provided in Section IV.

II. Related Work

The body of literature on RC on irreversible systems is large. Many approaches are focused on reversibility for debugging [30], allowing one to restore a program's state to any point in time by mixing checkpointing and forward traces [31] or by allowing executing inverse statements for non-destructive operations [32]. Inverse statements are typically automatically generated from machine code, or supported via interpreters [33]. This is the main difference in our approach: because we rely on MDE, we do not need to parse source or assembly code to generate reverse handlers. Rather, we rely on dedicated M2M transformations that allow us to invert the model's logic. The so-generated reverse model is then translated into source code relying on the same M2T transformation used for irreversible forward events.

In MDE, the reversibility of model transformations has been significantly studied. In [25], the authors introduce JTL, which is a specialised language that decouples the specification layer from the transformation layer of the metamodels. It relies on Answer Set Programming (ASP) [34] to account for non-bijective transformations that, when reversed, can generate multiple "source" models. ASP allows the retrieval of all valid models for a given transformation. GRoundTram [35], on the other hand, uses graph theory and functional programming to achieve reversibility. Similarly to JTL, this work relies on a specification language for the transformation, which is then used to generate forward and backward transformations. While these works have to deal with several RC-related hindrances, they do not apply MDE techniques to generate reversible code. Rather, they target the generation of M2M transformations.

In the context of PDES, the reversibility of event handlers has been studied extensively. After the seminal proposal in [15], several works attempted to automatically generate reverse handlers. In [18], the authors present LORAIN, which employs control flow analysis at compile time to generate LLVM Intermediate Representation code that allows for the reconstruction of the execution path taken in the forward code. In contrast, RAMSES [36] generates reverse events at runtime via software instrumentation. Any time that the simulation

[1]A persistent repository is available at https://doi.org/10.5281/zenodo.20120137.

state is updated (i.e., a destructive operation is carried out), a reverse memory write instruction is automatically generated and packed into a code buffer, which implements the reverse event—in a sense, this is an efficient way to implement incremental state saving [37] via reverse event execution.

Simulation models may use external libraries linked to the final model's executable. In this case, none of the above methods can generate complete reverse handlers, because the libraries cannot be directly reversed. This problem is addressed in [38], where external libraries are lazily disassembled and instrumented at runtime to generate reverse event handlers using a technique similar to [36].

In [39], [53], source-to-source transformation of C++ code is performed using the ROSE compiler infrastructure [40]. The initial version intercepts all memory-modifying operations, while a later optimised variant employs compile-time static analysis to exclude local variables from instrumentation, reducing the forward execution overhead. In both cases, the approach operates at the C++ AST level, recording information about performed updates in a data structure used to reverse their effects.

We take a path different from all the above works, as we work at a higher level of abstraction. In fact, we generate a reverse model starting from a forward model using dedicated M2M transformations. The final code is then automatically generated from a higher-level description, significantly simplifying the process and making it independent of the underlying computer architecture (as in [36]) or the compiler used (as in [18]). Since we have full control on the functional definition of the model (e.g., no opaque external libraries are allowed), we can perform transformations that effectively capture all the destructive operations to the model's state.

In [41], the authors propose a technique to generate reverse handlers for stochastic reaction diffusion simulations. The approach entails recognising the actual reaction executed in forward mode and executing the inverse logic. Given the simulation's special-purpose nature, this approach is hard to generalise to other domains.

III. Automatic Reverse Handler Generation

In PDES, irreversibility manifests in two distinct forms: data and structural. *Data irreversibility* occurs when the semantic value of a variable is destroyed (e.g., a destructive assignment or floating-point absorption [42]). While significant, it can be easily identified as it is confined to the application's internal scope. *Structural irreversibility* is more insidious: it arises when the application interacts with the execution environment (e.g., to allocate dynamic memory). In this Section, we illustrate how our M2M transformations manage a program implemented in the DESL language [23] to produce a reverse event handler from a forward event handler, while accounting for both forms of irreversibility.

A. The DESL Language

The DESL language [23] is structured around a set of abstractions designed to encapsulate the logic of generic DES models while maintaining platform independence. At the core of a DESL model is the `ClassDefinition` concept, which defines homogeneous classes of LPs. This abstraction allows modellers to define the behaviour of specific agents or entities within the system, using `EventHandlers` to implement the logic associated with a particular class. These handlers serve as the central mechanism for defining how LPs respond to simulation events, separating the behavioural logic from the underlying execution mechanics.

DESL allows implementing simulation handlers' logic using mbeddr [43], enabling us to treat handlers as structured models rather than opaque blocks of code. This is particularly useful because mbeddr allows the simulation developer to define event logic using a type-safe dialect of C. This aspect is particularly relevant, because the entire simulation logic is represented as an MDE model. This allows us to observe all destructive operations executed by a handler e, so as to generate the proper instrumentation in e_{instr} and the correct reversal logic in e^{-1}.

The language supports the manipulation of more complex and dynamic data structures with the `Collection` abstraction, which also enables object-oriented operations such as iteration via `foreach` statements and the addition or removal of elements by reference. The `Collection` asbtraction is also fundamental for our reversal approach: dynamic memory allocation/deallocation is controlled by our M2T trans formations only in relation to the logical operations carried out on this part of the language. This approach allows us to have full control on the interactions between the modeller and the heap, thus allowing us to simplify the management of heap reversibility.

To illustrate how a simulation model can be described in DESL, we provide in Figure 2 the reference implementation of the PHold benchmark [44]. This code will be used as one of the models used in the experimental assessment presented in Section IV.

B. Event Instrumentation and Reversal

To understand how we automatically generate reverse event handlers, we focus on the M2M transformation used to generate the reverse model e^{-1} from the forward model e, as depicted in Figure 1. The essential step is the inversion of reversible operations. This entails applying node transformations that map a forward operation to a backward operation, selecting the appropriate concepts. For example, a `BinaryOperation` representing a sum generates a `MinusOperation`, with operands mapped accordingly, as illustrated in Figure 3.

Performing this transformation at the model level ensures that the reverse model e^{-1} is fully formed before the M2T layer is invoked. This allows us to use the same M2T code-generation transformation for both the forward and reverse handlers, ensuring consistency across the generated simulation code.

1) Hierarchical Transformation Logic: Clearly, the ability to invert a single operation does not, on its own, produce a cor-

```
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);
```

Fig. 2: Implementation of PHold in DESL.

rect reverse event handler: it is just the fundamental building block. To obtain a correct and functional reverse handler, we must address a comprehensive structural transformation of the entire handler logic. An event handler is a composition of various control flow constructs (i.e., functions, loops, conditional branches) that must be reordered and transformed to undo the state changes performed during the forward execution. Our transformation process begins at the statement container level, for which the entire execution sequence must be mirrored. This means that the last statement executed in the forward path must be the first one executed during the execution of a reverse handler to execute a rollback operation.

The inversion of nested statement containers is handled at multiple levels of abstraction. At the macro level, a sub-container is a single unit mirrored within its super-container,

```
node<>
↪  invert_binary_expression(node<BinaryExpression>
↪  original)
{
    node<> inverted_node;

    /* * In a M2M approach, we instantiate a new node
    ↪  of the inverse concept. */
    if (original.operator == "+") {
        inverted_node = new node<MinusExpression>();
    } else if (original.operator == "-") {
        inverted_node = new node<PlusExpression>();
    }

    /* * Structural transformation: swap or map
    ↪  operands  to maintain semantic correctness
    ↪  during reversal. */
    inverted_node.left = original.left;
    inverted_node.right = original.right;

    return inverted_node;
}
```

Fig. 3: Determining an inverse operation

while its internal statements undergo their own mirroring. Control flow constructs require specialised inversion logic; for instance, a forward loop is transformed to reconstruct the iteration space in the opposite direction, reversing side effects in the exact inverse order of their occurrence.

Our M2M transformation orchestrates the generation of the reverse handler's body by creating a reordered list of statements. Declarations of local variables are extracted and placed at the beginning to ensure visibility. We also inject logic to restore the Random Number Generator (RNG) state—a critical requirement for repeatability in ROSS [45]—and remove early return statements, as reverse events must execute entirely to restore a consistent state. Finally, iterating over the reordered sequence, any non-destructive operation is automatically inverted to form the final reverse model.

2) Destructive Operations Management: As we discussed, not every operation is invertible, e.g. due to data irreversibility, such as assignments where the previous value is lost. In this case, the M2M transformations must instead inject nodes that allow for reconstructing the previously lost information. We formalise this in our instrumented metamodel by introducing the concept of *memento*. A memento is a modelling entity that concretises the entropy generated by a destructive operation. We introduce this concept in the model as a structural element to handle state restoration for destructive operations, rather than a low-level implementation detail.

To efficiently preserve the previous simulation states, our M2M transformation relies on a per-LP resizeable stack. During forward execution, the instrumented handler pushes mementoes onto the stack associated with the active LP. Clearly, if no rollback occurs, this stack would grow indefinitely. To reclaim stack memory, we must integrate the management of the mementoes stack with the *fossil collection* operation [6], i.e., the memory reclamation procedure proper to Time Warp.

To delimit the mementoes associated with a specific event, the M2M transformation injects logic to record the start and stop stack pointers within the `EventPayload` concept. Upon fossil collection, when events older than the Global

```
$CALL$[stackStateHandlingTemplate(
  variable -> node.supportVariable,
  stack -> "uIntStack"
)]

$IF$[node.thenPart.isNotEmpty]
  if ($REF$[node.supportVariable] & (1ULL <<
  ↪  $PropertyMacro[genContext.getStepObject(node,
  ↪  "baseBitIndex")])) {
  $CALL$[reverseStatementListTemplate(node.thenPart)]
  }
$ENDIF$

// Each Else-If gets the next sequential bit index
$LOOP$[node.elseIfs]
  // Calculate the index: base + 1 + loop_index
  // stored in a helper variable or macro for clarity
  if ($REF$[node.parent.supportVariable] & (1ULL <<
  ↪  $PropertyMacro[genContext.getStepObject(node,
  ↪  "baseBitIndex") + loop.index + 1])) {
   $CALL$[reverseStatementListTemplate(it.stmtLst)]
  }
$ENDLOOP$

$IF$[node.elsePart.isNotEmpty]
  if (($REF$[node.supportVariable] &
  ↪  $PropertyMacro[Helper.getMask(node)]) == 0) {

   ↪  $CALL$[reverseStatementListTemplate(node.elsePart)]
  }
$ENDIF$
```

Fig. 4: M2M reversal of conditional branches

```
{
  uint64_t $PropertyMacro[Helper.getUniqueName(node,
  ↪  "iter_cnt")] = 0;
  while ($COPY_SRC$[node.condition]) {
    $LOOP$[node.body.statements]
      $COPY_SRC$[it]
    $ENDLOOP$
    $REF$[Helper.getUniqueName(node, "iter_cnt")]++;
  }
  STACK_PUSH(content->uIntStack,
  ↪  $REF$[Helper.getUniqueName(node, "iter_cnt")])
}
```

Fig. 5: M2M instrumentation of loops

Virtual Time (GVT) are committed, all mementoes below the committed event's pointers in the per-LP stack are safely discarded. This allows the global stack to shrink dynamically, effectively bounding the memory footprint of the state-saving process.

3) Control Flow Topological Reversal: Beyond data irreversibility, conditional branches and loops present topological irreversibility. In reverse execution, a branch predicate or loop termination condition may be incomputable if the state variables it depends on were mutated.

For conditional branches, we record the traversed execution path as a *topological memento* (conceptually similar to a decision bit [45]). During forward execution, the M2M instrumentation, as shown in Figure 4, pushes bit-packed branch predicates onto a specialised `IfStack`, preventing pollution of the larger data memento stacks. The reversal transformation then pops this information to deterministically reconstruct the original control flow graph.

Similarly, for iterative constructs, the instrumentation transformation injects a transparent counter to capture the loop's depth. Upon termination, this iteration count is stored as a *cardinality memento* (see Figure 5). The corresponding

```
{
  uint64_t $PropertyMacro[Helper.getUniqueName(node,
  ↪  "loop_limit")];
  STACK_POP(content->uIntStack,
  ↪  $REF$[Helper.getUniqueName(node,
  ↪  "loop_limit")])
  // The Deterministic Reverse Loop
  {
    uint64_t
    ↪  $PropertyMacro[Helper.getUniqueName(node,
    ↪  "iter_idx")];
    for (
      $REF$[Helper.getUniqueName(node, "iter_idx")] =
      ↪  0;
      $REF$[Helper.getUniqueName(node, "iter_idx")] <
      ↪  $REF$[Helper.getUniqueName(node,
      ↪  "loop_limit")];
      $REF$[Helper.getUniqueName(node, "iter_idx")]++
    ) {
      $CALL$[reverseStatementListTemplate(node.body)]
    }
  }
}
```

Fig. 6: M2M reversal of loops

reversal transformation discards the potentially irreversible termination condition, as shown in Figure 6. Instead, it generates a deterministic loop that retrieves the cardinality memento and executes the inverted body the exact number of times observed during forward execution, ensuring accurate state reconstruction.

We note that, with the approach we have described so far, we reverse every statement (including data structure traversals). While this could be deemed inefficient compared to Incremental State Saving (ISS), which only cares about memory-modifying operations, this is a fundamental trade-off of the MDE approach. By reversing the logic at the statement level, we maintain a high degree of consistency, which is also required to properly reconstruct the heap as we shall discuss.

C. Handling Dynamic Memory

Relying on dynamic memory in the simulation model poses significant challenges for optimistic synchronisation, particularly due to the complex management of pointer-based structures and the non-deterministic nature of heap libraries. At the same time, modern simulation modelling paradigms offer significant expressive benefits if they can rely on non-flat linked data structures. Although DESL does not allow a direct invocation of `malloc`, it exposes dynamic data structures through the concept of `Collections`. Such concepts are internally realised using dynamic memory and linked data structures.

For this reason, our M2M transformations must be able to manage these data structures, so that reverse event handlers can reconstruct the heap consistently. The architectural pattern we describe here abstracts the implementation of `Collections` into a more general approach for reversible dynamic memory management in the case of MDE transformations. In this sense, our proposal can be used to generate correct reverse handlers independent of the `Collections` concept, thus also allowing an extension of DESL without requiring fundamental redesigns of the instrumentation/reversal transformation.

The major problem in making dynamic memory reversible is the behaviour of the `free` operation. When `free` is invoked, the system disconnects a semantic object from its virtual address. This is different from overwriting a variable, which results in a local, identifiable loss: deallocating memory gives the heap manager complete control and introduces randomness into the process. This behaviour is particularly critical when dealing with linked data structures, where the integrity of the overall data structure depends on the virtual addresses that link nodes together.

If a deallocated node is restored at a different address during a rollback, the integrity of the structure is compromised: any pointers held by other nodes would fail to point to the restored object, leaving the data structure in a corrupted state. Consequently, the mapping between a logical object and its virtual address is not a function of the simulation state alone, but also of the global state of the heap manager. Reversing a `free` operation by simply calling `malloc` is impossible without a historical record, as it attempts to extract order (the specific original address) from the disorder of the global pool.

Following the approach in [28], we overcome the non-determinism of standard allocators by introducing a custom dynamic memory manager that, for each LP, maintains a table of malloc areas, each responsible for managing contiguous memory blocks containing chunks of power-of-two sizes. The state of the chunks is tracked using a bitmap at the head of each block to minimise the metadata memory footprint. Whenever a call to `malloc` is executed, the allocator determines a suitable chunk size to serve the request and locates a free chunk by inspecting the bitmap. If no chunk is available, a new malloc area of the same size is initialised, keeping twice as many memory chunks as the original full one. Memory release is supported via a back pointer stored above an allocated chunk, which allows reaching the malloc area to which the chunk belongs. This custom allocator is injected into the final executable by the M2M transformations and initialised during simulation startup.

To support the reversibility of a `free` operation, we introduce the new primitive `remalloc(void *ptr)`, which bypasses the search for a free chunk in the bitmap and immediately marks the chunk as busy—the bitmap can be reached using the back pointer, as in a `free` operation. Since the relationship between a bit in the use bitmap and the chunk is bijective and static, this approach can effectively support reversing a `free` operation, avoiding potential inconsistencies in linked data structures.

It is important to note that the use of `remalloc` introduces a *strict temporal dependency* to avoid any possible collision in the allocation. For example, let us consider the sequence:

1) P = `malloc()` (Allocates slot 0x100)
2) `free`(P) (Frees slot 0x100)
3) Q = `malloc()` (Reuses slot 0x100)

If the reverse execution attempts to reverse step (2) via `remalloc(`P`)` before reversing step (3), a collision occurs: the slot 0x100 is currently held by Q. In our context, the temporal dependency is addressed by the M2M transformation: by executing the statements in an event handler in a mirrored fashion, we ensure LIFO ordering of the memory operations. Moreover, since a rollback operation in ROSS is supported by the execution of all event handlers from the current timestamp to the rollback point, the order is guaranteed also for cross-event allocation/deallocations.

The use of `remalloc` ensures *container consistency*, in the sense that during a rollback operation all nodes of the linked data structures exist at the right location. At the same time, it provides no guarantees about the container's content. This could lead to a *repopulation fallacy* [46], i.e. the assumption that simply restoring the structure of the heap is sufficient, because the (resumed) forward execution will repopulate the content of the buffers. The execution of an event, though, is only logically atomic, as it is constituted by a sequence of steps, which traverse *intermediate states*.

When a `free` is reversed using `remalloc`, the rest of the reverse event handler will expect to find in the buffers the values that were written during the forward execution. Conversely, by construction, `remalloc` does not reinitialise the memory, which (in case of buffer reuse as in the example above) will likely contain garbage. Any observation of that inconsistent state will cause the execution of the reverse handler to be incorrect. In other words, `remalloc` leaves the simulation state in a *zombie state*.

There are three possible strategies to handle this zombie state. The first is to log the memory content before executing `free`, and reinstall the previous content after a `remalloc`. This approach incurs a massive overhead in memory bandwidth and storage costs, effectively doubling the write traffic for every allocation. The second is to delay the actual release of the memory buffer until the execution of the `free` becomes committed (i.e., when the Global Virtual Time reaches the event). This approach has been employed by traditional reversible heap allocators for PDES (e.g., [28], [29]). While it solves the memory bandwidth problem, it may consume more memory than needed for model execution because the memory released by the model is not immediately available for subsequent allocations. Moreover, this approach somewhat clashes with the notion of reverse computation, because it checkpoints also values that could be recomputed by the reverse handlers.

The third approach entails decoupling the model's logic from the memory operations, employing a technique we refer to as *operand value logging*. During forward execution, any instruction that reads a value from the heap and uses it as an operand will push that value onto the memento stacks. During reverse execution, the reversed instructions will pop the mementoes from the stack and use them to perform the reverse actions. In this way, the heap container reinstalled by `remalloc` will remain (partially) in a zombie state throughout the execution of the reverse handler, and will be considered a *write-only* memory buffer. At the end of the reversal, any relevant value will be restored. This approach eliminates the need for expensive snapshots, ensures that `free`'d memory is reused by the model immediately, and guarantees correct

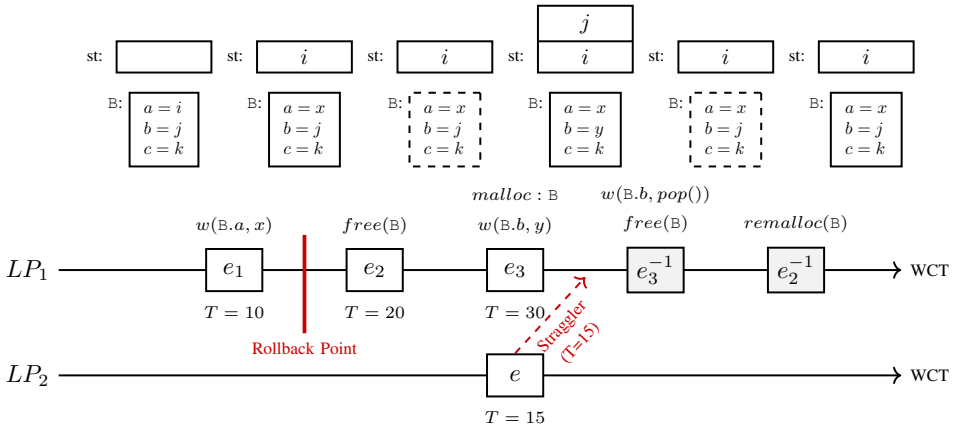

Fig. 7: Execution timeline illustrating dynamic state reconstruction outside the rollback interval. The diagram shows the evolution of a dynamic memory buffer `B` and the memento stack across forward and reverse events—when dashed, the buffer has been deallocated. Values updated prior to the rollback window (e.g., field a modified by e_1) physically survive the deallocation at e_2 thanks to the isolated per-LP custom allocator. This persistence allows the `remalloc` operation during reverse execution (e_2^{-1}) to perfectly restore the buffer's semantic state without requiring explicit logging of unread data.

control flow and data dependencies.

We emphasise that our dynamic state reconstruction approach is correct even in the presence of values written by events executed before the rollback point (i.e., values that are not read by rolled-back events and are therefore not present in the operand value stack). Consider the execution depicted in Figure 7. In this timeline, event e_1 at $T = 10$ updates field a of a buffer `B` with the value x. Subsequently, at $T = 20$, event e_2 executes a `free(B)` operation. Since e_2 does not access fields a or c during its execution, these values are not recorded in the operand value stack. At $T = 30$, event e_3 receives the same memory slot for a different allocation and modifies field b, pushing the previous value onto the memento stack. When a straggler message with timestamp $T = 15$ arrives from LP_2, a rollback is triggered. During the reverse execution, e_3^{-1} restores the previous value of b and releases the buffer, while e_2^{-1} invokes the `remalloc(B)` primitive. Importantly, the physical content of fields a and c is preserved throughout this process because our custom allocator segregates heaps per LP and maintains all allocation metadata in an external bitmap rather than within the buffers themselves.

Therefore, while classical reversibility theory normally considers the passive persistence of data in the heap as a factor outside the program's control, our construction of per-LP allocators ensures that previous values survive even if a buffer is deallocated. The zombie state, in this view, is not physical corruption but a controlled logical transition: data integrity is guaranteed by the stability of the segregated system, making the restoration operation intrinsically correct and non-probabilistic[2]. It is therefore possible to argue that the redundancy of operand value logging and the reversal of local variables does not protect the survival of data in the heap (which is guaranteed by the allocator's structure itself), but rather allows us to determine the proper control flow during the rollback. Overall, the steps taken to reconstruct a previous consistent state are summarised in Table I.

[2]This design principle is logically congruent with the philosophy of Dancing Links (DLX) in Knuth's Algorithm X [47], where an element's removal from a system does not imply the destruction of its internal information, but merely its temporary exclusion from the active logic.

D. Floating-Point Arithmetic

Floating-point arithmetic introduces another class of irreversibility, particularly relevant in PDES, given the high incidence of numerical operations in typical simulations. This irreversibility stems from the approximate nature of floating-point operations.

To simplify the discussion, let us consider only floating-point addition ($\oplus$). Differently from integer arithmetic, floating-point addition is not an abelian group: it lacks associativity and, more importantly, invertibility. In fact, if we consider $x' = x \oplus c$, the operation $x = x' \ominus c$ does not necessarily restore x to the previous value. This is due to absorption (where small values added to large values are shifted out of the mantissa) and rounding modes.

One approach to address this issue and make floating-point operations reversible is to estimate the round-off error introduced by a floating-point operation. For example, the 2Sum algorithm [48] allows computing the approximate result $x \oplus c$ and the round-off error $E = x+c-(x \oplus c)$. One might propose logging E to enable reversibility. However, this approach fails for multiple reasons. First, the error term E is itself a floating-point number requiring the same storage width (e.g., 64 bits) as the original value x. Logging E saves no space compared to simply logging the original x. Second, even if E is available, the inverse operation must also be evaluated using floating-point hardware, which introduces additional error. A perfect reconstruction would require arbitrary precision arithmetic, which is computationally prohibitive.

TABLE I: Management of dynamically allocated data during the rollback of a `free` operation (event e_2 in Figure 7).

Operation on data	**Action in forward event e_2**	**Data preservation**	**State after rollback**
Data read by e_2	Pushed onto stack (Operand logging)	Memento Stack	Available in reverse handlers, not necessarily in heap (local variable)
Data overwritten by e_2	Old value saved (Destructive assignment)	Memento Stack	Correctly restored in heap
Data ignored by e_2	No logging action	Passive heap persistence (LP isolation)	Physically preserved (Controlled logical transition)

TABLE II: SLOC Count for different model versions.

Model name	**DESL version**	**Generated C code**
PHold	50	233
PCS	340	1113

Consequently, we treat all floating-point operations as destructive assignments. To maintain type alignment, we partition the logging mechanism discussed in Section III-B into multiple stacks rather than a single generic buffer: a `double` stack for floating-point operands, and an `uint64` stack for integers and pointers.

IV. Experimental Assessment

To evaluate our proposal, we employed two complementary benchmarks: the synthetic PHold model illustrated in Figure 2, which provides a controlled workload to isolate scheduling overhead with minimal LP state, and a high-fidelity Personal Communication System (PCS) that exercises large LP states, floating-point computation, and dynamic memory allocation. PCS models a mobile network in accordance with the specifications in [49]. Each LP simulates a hexagonal network cell, with events associated with call initiation and termination. As simulated devices traverse the coverage area, they require handovers across distinct LPs, potentially leading to rollbacks. For every active call, the model allocates dynamic data structures of different sizes to manage channel allocation and configure call parameters, ensuring the required Signal-to-Noise Ratio (SNR) is maintained.

Since our primary contribution lies in the MDE domain, we quantify the concrete reduction in development effort afforded to the simulation modeler. To this end, we compare the Source Lines of Code (SLOC) required to implement a model in the DESL language with the volume of C code automatically generated by our M2M transformations—all generated code is functional and exercisable. This metric effectively illustrates the "compression ratio" of a DESL model while enabling the use of performance-competitive support for PDES execution. The results, computed using the standard utility `sloccount` [50], are reported in Table II. From the results, we observe that the abstraction gap provided by DESL yields a significant reduction in implementation verbosity. Specifically, the PHold model demonstrates an expansion factor of approximately $4.7\times$, while the more complex PCS model exhibits a factor of roughly $3.2\times$. This expansion factor can be interpreted as a quantitative measure for the cognitive load alleviated from the modeller: our approach automates the generation of nearly 70% of the final codebase, encompassing low-level simulation mechanics and synchronisation protocols that typically constitute a primary source of implementation errors in PDES.

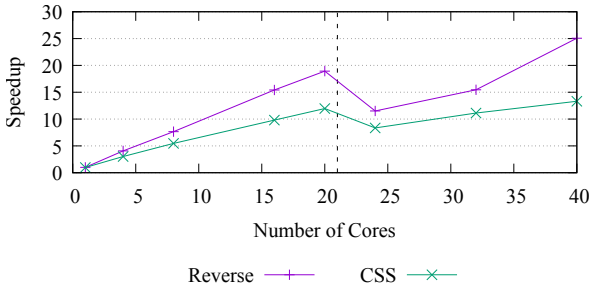

Fig. 8: Speedup over sequential execution for PCS when varying core count. The dashed line indicates when hyperthreads are first used, resulting in an initial performance drop.

To assess the runtime efficiency of our proposal, we compare the speedup of the PCS benchmark automatically generated from the DESL specification against a native, hand-optimised C implementation on ROSS relying on Copy State Saving (CSS), as previously used in [23]. We selected CSS as the baseline because it represents the standard rollback mechanism for PDES developers who do not manually write reverse event handlers. A direct comparison with other automated approaches such as LORAIN [18], RAMSES [36], or the ROSE-based solution [39], [53] is not feasible, as they are tied to specific compiler infrastructures and no public benchmark infrastructure exists for head-to-head evaluation on identical workloads.

We have run our experiments on a machine equipped with an Intel Xeon Silver 4210R CPU, with 20 physical cores (40 hyperthreads) running at 2.40GHz, and 160 GB of RAM. The machine has two NUMA nodes. In this experiment, we have used 22,500 LPs, with a load factor on the mobile network of 50% over 1,000 channels per cell—this means that, on average, 500 nodes are present in each Collection used by the model, i.e. two Collections per LP. New calls are injected into the system according to an Erlang distribution with a mean of 0.16, and each call has an average duration of 120 seconds (drawn from an exponential distribution). We have run these high-load simulations for a total of 10 simulated minutes. Each configuration was executed 10 times with varying random seeds.

The results of the speedup over sequential execution are

TABLE III: Memory usage in extreme configurations.

Core Count	Reverse	Checkpointing
2	3.74 GB	6.7 GB
40	9.59 GB	34.67 GB

reported in Figure 8 when varying the number of cores used to execute the simulation. As a first observation, we see performance degradation as the core count exceeds 20: this is expected, as at 21 cores the system starts relying on hyperthreads, which is detrimental to compute-bound workloads like PDES. In particular, the microarchitectural side effects (e.g., L1 cache sharing between two hyperthreads) on a reduced number of cores introduce skew in the logical time of the LPs, leading to an increased number of rollbacks in LPs run by threads using a physical core in isolation.

Overall, we observe that simulations using reverse computation achieve consistently higher speedups than those relying on traditional state saving. This is a result consistent with the literature, as in the considered model the rollback length is relatively short while the state size is large. The events, at the same time, operate on small portions of the state, reducing the amount of information saved on the stacks compared to full checkpoints. At the same time, the significance of this result lies in the implementation method: we achieve these speedups through fully automated code generation, thus eliminating the high engineering costs and defect risks typically associated with manually implementing reverse event handlers.

To verify the correctness of the automatically generated reverse handlers, we have implemented the COMPADS model [54] and used it to compare the final simulation state produced by the optimistic parallel execution against the state produced by a sequential run. Across all configurations and all 10 independent runs, the two execution modes produced identical results, confirming that our M2M transformations preserve the causal correctness of the simulation.

To complete our experimental assessment, we have studied the memory footprint of the simulation using the extreme parallel configurations (i.e., 2 cores and 40 cores), which show the smallest and largest differences in speedup, respectively, between the two compared solutions—we have not considered the single-core baseline, as it relies on a sequential scheduler and causes no rollbacks. We report in Table III the peak memory usage by the two rollback mechanisms. As expected, checkpointing incurs higher memory usage, up to $3.6\times$ that of the reverse computation. This is intrinsic to state saving, as all buffers associated with call records are copied after each event, even when the actual state update is sparse or non-present in forward execution. The reduced time to reinstall the consistent checkpoint in case of a rollback does not pay off, especially because the number of rolled-back events is reduced.

At 2 cores, despite the disparity in memory footprint, the performance of the two rollback methods converges. This is related to memory bandwidth availability: with a small thread count, the system's memory bus is not saturated and can therefore handle the heavier state-saving operations without stalling the CPU. Consequently, the latency penalty is effectively hidden by the memory controller's high throughput, while at higher core counts we empirically observe the effect of the memory wall [51], [52].

V. Conclusions

In this paper, we have presented an MDE-based approach to automatically generate reverse event handlers for optimistic PDES. By leveraging M2M transformations within the DESL language, we demonstrated that the historical complexity of manual reverse handler implementation can be effectively mitigated. Our methodology transparently handles also dynamic simulation states through the introduction of a reversible memory allocator, ensuring consistency in speculative execution environments. The correctness of the generated reverse handlers was verified through the COMPADS model, confirming identical results between optimistic parallel and sequential execution. Empirical results validated that this automated approach preserves and improves the performance of hand-coded models while enhancing reliability and maintainability.

Acknowledgements

This paper has been partially supported by the European Union—Next Generation EU, Mission 4, Component 2, CUP E53D23008200006 (Domain).

References

[1] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[2] R. Ronngren and M. Liljenstam, "On event ordering in parallel discrete event simulation," in *Proceedings Thirteenth Workshop on Parallel and Distributed Simulation*. IEEE Comput. Soc, 2003, pp. 38–45.

[3] K. Perumalla *et al.*, "Computer science research needs for parallel discrete event simulation (PDES)," U.S. Department of Energy, Tech. Rep. 1855247, 2022.

[4] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, 1979.

[5] D. M. Nicol, "The cost of conservative synchronization in parallel discrete event simulations," *Journal of the ACM*, vol. 40, no. 2, pp. 304–333, 1993.

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

[7] J. S. Steinman, "Breathing time warp," *Simuletter*, vol. 23, no. 1, pp. 109–118, 1993.

[8] M. Ianni, R. Marotta, D. Cingolani, A. Pellegrini, and F. Quaglia, "The ultimate share-everything PDES system," in *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 2018, pp. 73–84.

[9] P. Andelfinger, T. Köster, and A. M. Uhrmacher, "Zero lookahead? zero problem. the window racer algorithm," in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 2023, pp. 1–11.

[10] A. Piccione, P. Andelfinger, and A. Pellegrini, "Hybrid speculative synchronisation for parallel discrete event simulation," in *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 2023, pp. 84–95.

[11] M. P. Frank, "Introduction to reversible computing: motivation, progress, and challenges," in *Proceedings of the 2nd conference on Computing frontiers*. ACM, 2005, p. 385.

[12] K. S. Perumalla, *Introduction to reversible computing*. Whittles Publishing, 2013.

[13] M. V. Zelkowitz, "Reversible execution," *Communications of the ACM*, vol. 16, p. 566, 1973.

[14] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM Journal of Research and Development*, vol. 5, pp. 183–191, 1961.

[15] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, "Efficient optimistic parallel simulations using reverse computation," *ACM Transactions on Modeling and Computer Simulation*, vol. 9, no. 3, pp. 224–253, 1999.

[16] E. Cruz-Camacho and C. D. Carothers, "The needle in the one-billion event-haystack: A simple method for checking reverse handlers in parallel discrete event simulation," in *Proceedings of the 2025 Winter Simulation Conference*. IEEE Computer Society Press, 2025.

[17] K. S. Perumalla and R. M. Fujimoto, "Source code transformations for efficient reversibility," Georgia Institute of Technology, Tech. Rep. GIT-CC-99-21, 1999.

[18] J. M. LaPre, E. J. Gonsiorowski, and C. D. Carothers, "LORAIN: a step closer to the PDES 'holy grail'," in *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 2014, pp. 3–14.

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

[20] P. Wilsdorf *et al.*, "A model-driven approach for conducting simulation experiments," *Applied sciences (Basel, Switzerland)*, vol. 12, no. 16, p. 7977, 2022.

[21] S. Zschaler and F. A. C. Polack, "Trustworthy agent-based simulation: the case for domain-specific modelling languages," *Software & Systems Modeling*, vol. 22, no. 2, pp. 455–470, 2023.

[22] R. Marotta and A. Pellegrini, "Model-driven engineering for high-performance parallel discrete event simulations on heterogeneous architectures," in *Proceedings of the 2024 Winter Simulation Conference*. IEEE, 2024, pp. 2202–2213.

[23] 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*. ACM, 2025, p. 12.

[24] V. Pareek, S. Gupta, S. Jain, and D. Jain, "Fault tolerant and testable designs of reversible sequential building blocks," in *2014 International Computer Science and Engineering Conference (ICSEC)*. IEEE, 2014, pp. 452–457.

[25] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "JTL: A bidirectional and change propagating transformation language," in *Software Language Engineering*. Springer, 2011, pp. 183–202.

[26] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *Theory and Practice of Model Transformations*, ser. LNCS. Springer, 2009, pp. 260–283.

[27] C. Carothers, D. Bauer, and S. Pearce, "ROSS: A high-performance, low-memory, modular time warp system," *Journal of parallel and distributed computing*, vol. 62, no. 11, pp. 1648–1669, 2002.

[28] R. Toccaceli and F. Quaglia, "DyMeLoR: Dynamic memory logger and restorer library for optimistic simulation objects with generic memory layout," in *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*. IEEE CS, 2008, pp. 163–172.

[29] A. Pellegrini, R. Vitali, and F. Quaglia, "Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects," in *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation*. IEEE, 2009, pp. 45–53.

[30] I. Lanese, "From reversible semantics to reversible debugging," in *Reversible Computation*, ser. LNCS. Springer, 2018, pp. 34–46.

[31] D. Z. Pan and M. A. Linton, "Supporting reverse execution for parallel programs," *SIGPLAN Notices*, vol. 24, no. 1, pp. 124–129, 1989.

[32] T. Akgul and V. J. M. Iii, "Assembly instruction level reverse execution for debugging," *ACM Transactions on Software Engineering and Methodologies*, vol. 13, pp. 149–198, 2004.

[33] B. Biswas and R. Mall, "Reverse execution of programs," *ACM SIGPLAN Notices*, vol. 34, pp. 61–69, 1999.

[34] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance," *Communications of the ACM*, vol. 54, no. 12, pp. 92–103, 2011.

[35] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano, "GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2011, pp. 480–483.

[36] D. Cingolani, A. Pellegrini, and F. Quaglia, "RAMSES: Reversibility-based agent modeling and simulation environment with speculation-support," in *Euro-Par 2015: Parallel Processing Workshops*. Springer, 2015, pp. 466–478.

[37] D. West and K. Panesar, "Automatic incremental state saving," in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*. IEEE, 1996, pp. 78–85.

[38] D. Cingolani, A. Pellegrini, M. Schordan, F. Quaglia, and D. R. Jefferson, "Dealing with reversibility of shared libraries in PDES," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 2017, pp. 41–52.

[39] M. Schordan, D. Jefferson, P. Barnes, T. Oppelstrup, and D. Quinlan, "Reverse code generation for parallel discrete event simulation," in *Reversible Computation*, ser. LNCS. Cham: Springer International Publishing, 2015, pp. 95–110.

[40] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Proceedings of the 2011 Cetus Users and Compiler Infastructure Workshop*, 2011.

[41] Z. Lin and Y. Yao, "Parallel discrete event simulation of stochastic reaction and diffusion using reverse computation," in *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*. IEEE, 2015.

[42] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, 1991.

[43] 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*. ACM, 2012, pp. 121–140.

[44] R. M. Fujimoto, "Performance of time warp under synthetic workloads," in *Distributed Simulation*. San Diego, CA, USA: Society for Computer Simulation International, 1990, pp. 23–28.

[45] C. Carothers, D. Bauer, and S. Pearce, "ROSS: a high-performance, low memory, modular time warp system," in *Proceedings 14th Workshop on Parallel and Distributed Simulation*. ACM, 2000, pp. 53–60.

[46] C. H. Bennett, "Logical reversibility of computation," *IBM Journal of Research and Development*, vol. 17, pp. 525–532, 1973.

[47] D. E. Knuth, "Dancing links," arXiv [cs.DS], 2000.

[48] O. Møller, "Quasi double-precision in floating point addition," *BIT Numerical Mathematics*, vol. 5, no. 1, pp. 37–50, 1965.

[49] S. Kandukuri and S. Boyd, "Optimal power control in interference-limited fading wireless channels with outage-probability specifications," *IEEE Transactions on Wireless Communications*, vol. 1, pp. 46–55, 2002.

[50] D. A. Wheeler, "More than a gigabuck: Estimating GNU/linux's size," Tech. Rep., 2001.

[51] W. A. Wulf and S. McKee, "Hitting the memory wall: implications of the obvious," *Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[52] S. McKee, "Reflections on the memory wall." ACM Press, 2004, p. 162.

[53] M. Schordan, T. Oppelstrup, D. R. Jefferson, P. D. Barnes, and D. Quinlan, "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*. ACM, 2016, pp. 111–122.

[54] T. Köster, A. M. Uhrmacher, and P. Andelfinger, "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*. ACM, 2022, pp. 31–32.