Proceedings of the 2023 Winter Simulation Conference
C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.

RUSTSIM: A PROCESS-ORIENTED SIMULATION FRAMEWORK FOR THE RUST LANGUAGE

Kevin Frez
Mauricio Oyarzún

Facultad de Ingeniería y Arquitectura
Universidad Arturo Prat
Av. Arturo Prat 2120
Iquique, CHILE

Alonso Inostrosa-Psijas

Escuela de Ingeniería Informática
Universidad de Valparaíso
General Cruz 222
Valparaíso, CHILE

Francisco Moreno

Depto. de Matemática y Ciencia de la Computación
Universidad de Santiago
Av. Bernardo O'Higgins 3363
Santiago, CHILE

Gabriel Wainer

Dept. Of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa ON K1S 5B6, CANADA

ABSTRACT

We present RustSim, a library for discrete-event process-oriented simulations designed and implemented in the Rust programming language. It includes a broad set of classes to allow the user to implement simulation processes and process-oriented primitives. The flexible modular design of RustSim allows users to extend its functionality. In addition, RustSim includes mechanisms to avoid inconsistencies when applying state-changing primitives that other libraries in the language's ecosystem do not provide. We take advantage of Rust generators (coroutine equivalents) to implement process-oriented simulation primitives. Finally, the library's internal process handling structure is discussed in detail, including its implementation, how simulations are executed, and a case study with a highly detailed example of its use.

1 INTRODUCTION

Discrete-Event Simulation (DES) has long been proven to be an effective and powerful tool that allows studying and understanding the behavior of complex discrete dynamic systems. DES is a computing program written in a programming language that emulates the behavior of a physical or real system. DES is advantageous when the studied system is under design or when (due to its nature) the operational costs make it impossible to collaborate directly with it (Wainer 2017). In DES, a physical system corresponds to a set of entities collaborating to accomplish a global objective. A set of state variables represents the states of the physical system and is used to describe the system state at any time (Cassandras and Lafortune 2008). The processing of discrete timestamped events advances simulation time and generates changes to the state variables, representing changes in the physical system’s state.

Process-Oriented Simulation (POS) is a type of DES in which the simulation model can be conceptually regarded as “processes” that use (and eventually compete for) the system’s “resources.” The simulation model is represented as a set of concurrently executing processes. In POS, special simulation primitives are

needed to allow processes to change among states (Dahl and Nygaard 1966) and to allow time to advance (Schwetman 1986).

This article presents a new process-oriented simulation library developed in the Rust programming language. Rust is a modern programming language focusing on safety, speed, and concurrency. It offers a unique set of features, including a robust type system, memory safety guarantees, and built-in support for concurrency and parallelism. The addition of generators, a feature introduced in Rust 1.48, allows developers to write more efficient and expressive code. While still a relatively new feature, Rust's generators have the potential to significantly enhance the language's ability to handle asynchronous programming and complex data processing tasks. Our proposal for a POS library for Rust provides a reduced set of easy-to-learn primitives to ease programmer efforts when developing a POS model. A case study is presented to introduce the reader to the use of the library.

The remainder of this article is structured as follows: Section 2 explains the foundations of POS and recalls what a coroutine is and its importance in it. Then, Section 3 is devoted to present works related to this proposal. Next, in Section 4, we present our library and provide details about its design, implementation, and a case study of its use. Finally, Section 5 ends with the main conclusions of this work.

2 BACKGROUND

2.1 Coroutines

A coroutine is defined as an object function that can suspend its execution and resume it at the point of its suspension while keeping the state of all its local variables (Helsgaun 1999). To support this, the coroutine has its own stack of procedure activations. Thus, a coroutine may suspend itself, and another may resume its execution. This coroutine execution sequencing is called alternation and makes it suitable to support POS, and it has been used for this purpose since SIMULA (Dahl and Nygaard 1966). Most programming languages now have support for coroutines, such as C++20, Python since version 2.2 and Rust support, which is still experimental, as discussed in the official documentation (last accessed on March 3rd, 2023).

2.2 Process-Oriented Simulation

POS is one of the three approaches to DES identified early in the 1960s by (Lackner 1964). According to the POS worldview, the simulation model can be conceptually regarded as "processes" that use the system's "resources" (Schwetman 1990). The simulation model is represented as a set of concurrently executing processes. Processes are represented as a set of activities in a specific order that need resources to be performed, and the model tracks the status of each process as it progresses through these activities. At each time, any of the processes can be in one of three mutually exclusive states: active, holding, or passivated (Saydam 1985). The active state indicates that the process is currently being processed. The holding state indicates that the process will produce a delay in simulated time (thus advancing the simulation time); it is often used to represent that the process is performing a simulated task. Finally, the passivated state means that the process is currently inactive and is waiting for an event to occur to be reactivated.

Given the above-mentioned states a process may be in, the SIMULA language (Dahl and Nygaard 1966) defined a set of primitives to manipulate processes to switch among states. To implement POS in a general-purpose programming language, special support is required (in the form of coroutines or generators) to allow the implementation of POS simulation primitives (Marzolla 2004). At least the following primitives are needed:

- **Activate:** allows one process to activate another process in passive state.
- **Passivate:** allows a process to passive itself.
- **Hold:** allows a process to produce a delay, thus advancing simulation time.

Despite being more challenging to learn than the other approaches, once mastered, POS is more suitable for simulating large models with several types of entities (Marzolla 2004). The process-oriented approach to DES is well-suited for large and complex modeling interactions between processes, such as manufacturing systems, supply chains, and logistics networks. It allows modelers and researchers to model and analyze the behavior of individual processes and interactions between them, providing a detailed and accurate representation of the system (Cassandras and Lafortune 2008).

2.3 The Rust Language

Rust is a statically typed, compiled programming language that runs on various platforms (Klabnik and Nichols 2018). It is known for its strong emphasis on safety, concurrency, and performance. Rust has a borrow checker that ensures memory safety and data-race freedom, eliminating many vulnerabilities and bugs that plague other system languages. It uses a lightweight threading model and it supports asynchronous programming, making it easy to write concurrent and parallel code that can take advantage of multiple cores. Rust is designed to be fast and efficient, with low-level control over system resources and support for advanced optimization techniques. It has a robust static type system that helps to catch errors at compile-time, making it easier to write correct and reliable code. Rust has a clean and expressive syntax, focusing on productivity and ease of use. It also has a large and active community that maintains a rich ecosystem of libraries and tools. The critical principle of Rust's type system is exclusivity: data can either be mutably accessed through a single owner or shared immutable among many references, but not both simultaneously. This behavior is defined by the concepts of "ownership," "borrowing," and "lifetimes." By enforcing these concepts statically, Rust prohibits shared state mutation, which allows many memory-related issues to be detected at compile-time. Problems such as double frees, use of freed memory, reading pointers to invalid locations, or invalidating iterators are impossible situations in Rust. Overall, Rust is a versatile and powerful programming language well-suited for a wide range of applications.

3 RELATED WORK

Initially, POS models were implemented in specific languages for this purpose, such as Simula (Dahl and Nygaard 1966) or GPSS (Gordon 1978), to name a few exponents. More recently, POS libraries have been developed on top of general-purpose high-level programming languages.

The work of (Marzolla 2004) presents a C++ library for POS called libcppsim. The library is based on coroutines for the abstraction of concurrently executing simulated processes. It provides a main simulation entity process, and the process scheduling primitives are based on those from the SIMULA (Dahl and Nygaard 1966) programming language, through methods that allow to suspend the active process for a certain amount of time, schedule the activation of another process in the future or to cancel a pending process. Since C++ (at the time of libcppsim development) did not have native support for coroutines, libcppsim relies on a portable third-party implementation. The library incorporates facilities for statistics calculation and abstract templates for pseudo-random number generation based on the MRG32k3a algorithm. This library has been successfully used in simulations of large-scale software services such as web search engines (Marin et al. 2019; Marín et al. 2017; Gil-Costa et al. 2013) and NoSQL database system implementations (Ovando-Leon et al. 2019). One drawback is that it has not received updates since 2013 (Marzolla 2013).

SimPy (SimPy 2017; SimPy 2007) is a widely used library for performing POS. It is developed on top of the standard Python language and provides support for a series of high-level constructs to ease the user with frequently used simulation tasks. For the simulation of processes, SimPy relies on Python generators, which force the user to explicitly call yield statements to pause functions representing the simulated process or to activate others. Thus, for the simulation of large process models, the user must handle many yield statements to correctly coordinate the simulated processes, which may be difficult. Simulus (Liu 2020) is a Python-based discrete-event simulator that also supports the process-oriented worldview. Similarly to SimPy, it provides several high-level constructs to help the modeler with common simulation tasks. Unlike

SimPy, which uses generators to support the process-oriented worldview, Simulus uses coroutines from a C-extension called Greenlet (Rigo and Tismer 2021) for implementing simulated processes. It also allows the parallel simulation (on shared or distributed memory platforms) of models using the YAWNS conservative synchronization protocol (Nicol 1993) and MPI (through the MPI4Py package) for the message-passing communication between computing nodes.

SimRs is a general-purpose library for conducting simulations that provides various mechanisms such as a scheduler, state management, and queues. It is written in Rust and it provides the necessary types for implementing simulation models and includes a Component interface for defining the behavior of entities within the simulation. This includes specifying the events they emit and how they are added to the list of future events. While offering a high degree of flexibility, developing models with SimRs can be less intuitive. To create a model in SimRs, a new structure must be defined for each entity participating in the simulation. These structures must implement the Component interface and define the logic that the entity will execute, and the event required for scheduling. When it is time for the entity to run, the event with which it was scheduled can be examined to alter the entity's behavior, or it can be ignored, and the same logic can be consistently applied. The events do not self-stack the list of future events, requiring the user to manually manipulate the list of future events. It does not provide the ability to pause and resume functions, requiring the function to return and reset its state to execute another event. The State structure can store and retrieve data and mitigate the loss of information or facilitate the exchange of data between functions. However, this may complicate the modeling of complex processes involving the interweaving of multiple actions. Overall, SimRs has the following limitations:

- SimRs is not process-oriented, with no relationship between emitted events and modifying the list of future events.
- SimRs cannot pause and resume functions.
- Modeling complex processes can be more challenging due to the increased code required for synchronizing actions between functions.

Desim is a library that implements a discrete event simulation framework inspired by the Python library SimPy, written in Rust and it uses an experimental generator functionality in the Rust language to provide a better developer experience. To create a simulation model in Desim, a function must be made for each entity model to be studied. This function defines the logic that the entity will execute when it is run and returns an instance of a generator containing this logic. This generator will have the ability to give control of the execution back to the simulation every time it emits an event, and the next time it is run, it will continue from the same point where it previously gave control. As a constraint, all entities must emit the same event, which in this case is an enumeration with many variants that the process manager can respond to with a specific action depending on the obtained variant by modifying the list of future events and the simulation's state variables. Desim uses experimental coroutine generators provided by Rust only as of the language compiler's RFC 2033 (07/2017) and requires its users to enable the generator's functionality. This version offers six operations specified in its documentation: Timeout (time), Event (event, time), Request (resource), Release (resource), Wait, and Trace (event used only for logs, reschedules the generator immediately). Desim links events to modify the list of future events. The developer cannot access the Future Event List (FEL) but can modify it indirectly by emitting events with the entities. It provides six events to emit but does not allow multiple entities to be activated with a single event, and activating an event means stopping the one that emits it. The way Desim associates events with modifying the list of future events has some deficiencies: When an entity emits an event to activate another entity implicitly, it is deactivated until it is activated again. Suppose entities are allowed to self-schedule by emitting events. Eventually, it will result in only one entity being active at any given time, which goes against the quasi-parallelism that the process-oriented approach seeks to provide.

4 RUSTSIM

RustSim is a general-purpose process-oriented simulation library written in the Rust language. It is based on some of the primitives previously defined in SIMULA (Dahl and Nygaard 1966). RustSim implements the primitive scheduling functions hold(), activate(), passivate(), and cancel() to allow modification of the state of the processes (which alternate states between Active, Suspended, Passive, and Terminated). For simplicity, RustSim only implements the states Active and Passivated. We will understand by Active state when an entity has an associated event in the FEL and Passive otherwise.

Although not explicitly defined by the library, the Terminated state is represented by the Completed state of some generators in Rust. Thus, resuming a generator that has reached the Terminated/Completed state will result in the early termination of the program execution by the library. The library will also interrupt the simulation execution if it detects an abnormal situation involving invalid operations concerning the state in which they are (i.e., as activating an already active entity or passivating a passivated entity). Since the library operates sequentially, the Suspended state loses usefulness since if the generator executes its logic, no other part of the code will run.

In RustSim, the FEL is implemented using a priority queue (heap) to store events chronologically. The FEL enables efficient processing of such events during the simulation and ensures they are executed at scheduled times.

4.1 Classes and Structures

As we can see in Figure 1, the simulation library comprises several interrelated classes used to conduct a simulation. Following is a brief description of these classes:

- **Container:** This class is responsible for storing user-written generators. It creates and returns a Key that is associated with the entered generator. Accessing a generator requires using the associated Key. In addition, the Container maintains the generator's current state (Passive, Active).
- **Scheduler:** This class is responsible for storing EventEntries associated with the generators. It maintains the current time of the simulation. When an EventEntry is removed from this structure, the simulation time advances. The Scheduler always returns the EventEntry closest to the current simulation time.
- **EventEntry:** This class represents a discrete event in the simulation. It comprises a duration representing the simulation time when the event occurs and a Key associated with the generator that emits the event.
- **Key:** This structure allows for identifying a generator. It is used in Container to access the associated generator and in Scheduler to associate an EventEntry with the generator that emits it.
- **EntityState:** This enumeration has two variants (Passive and Active) that emulate the states in which processes can be found in SIMULA, but simplified, reducing the number of states from four to two.
- **ShouldContinue:** This enumeration has two variants (Advance and Break) that indicate whether a simulation step was processed successfully or if the simulation could not advance because no events are stored in the Scheduler.
- **GeneratorState:** This enumeration has two variants (Yielded and Complete) that come directly from Rust's generators. When a generator pauses its execution, it does so by yielding with one of the values of Action and returns Yielded(Action). However, if it terminates its execution completely (i.e., it returns, instead of yielding), it returns the Complete variant with the value returned from the generator inside it.
- **Action:** This enumeration has five variants (Hold, Passivate, ActivateOne, ActivateMany, and Cancel) used to emulate the primitives used in SIMULA. It is a logical error to perform Hold-Passivate-Activate or receive a Cancel if the generator is in a Passive state or receive an Activate if the generator is in an Active state.

- **Simulation:** This structure is designed to be the user's point of contact with the library. It is the composition of the Container and Scheduler. It provides all the capabilities to execute the user's simulation model.

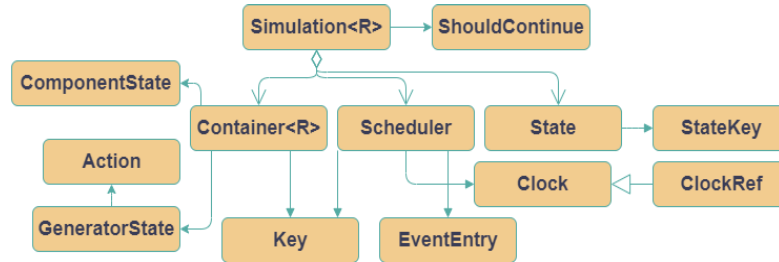

Figure 1: Class diagram of the RustSim simulation library.

We can separate the structures into two groups: public and private. In the first group, we find the structures the user can interact with (Simulation, EntityState, Key, Action, and EventEntry). In the second group, formed by Container and Scheduler, we find the ones that the user cannot access directly.

4.2 Simulation Process

To run simulation models, RustSim requires that the entities participating in the simulation be configured in separate functions. Each entity becomes an isolated environment with its own available data and no direct access to the data of other entities. Should it be needed to share and manipulate data from multiple entities, the data can be saved in the State (associated with a key), and the StateKey can be delivered to the respective functions as input parameters. In a simulation, generators are added using the add_generator_fn() method of the Simulation class (Figure 2). This method stores the generator in a Container and generates a key associated with the generator, which is returned to the function invoker. Then, using the previously generated key, the generator is scheduled at a particular time during the simulation using the schedule() or schedule_now() methods. These methods internally call the analog methods of the Scheduler class, which will insert an event associated with the key at the specified simulation time. This process is repeated for each entity involved in the simulation.

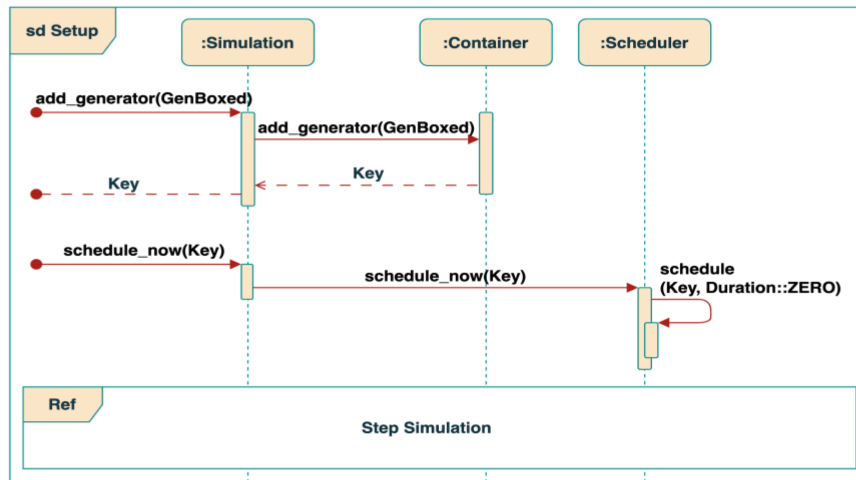

Figure 2: Sequence diagram to configure RustSim simulation library.

When executing a simulation step, the step_with() method of the Simulation class is invoked (Figure 3). Depending on the result of this execution, the method will return one of the ShouldContinue values (Break or Advance). To do this, Simulation invokes the pop() method of the Scheduler class, which attempts to extract an event (Figure 3, section 1). If it is impossible to extract an event, the Scheduler returns an Option::None, and Simulation returns a ShouldContinue::Break, indicating that it could not process an event and the simulation execution ends (Figure 3, section 2). However, if an event is successfully extracted, an EventEntry is returned. The current simulation time is updated to the timestamp of the extracted event, and an Option::Some with the EventEntry inside is returned to the Simulation class. Then, from EventEntry, the key associated with the event is extracted, and the step_with() method of Container is invoked, which identifies the generator associated with the key and resumes its execution using the resume() method (which is a native Rust method that all generators have). Finally, the execution results in the generator's state, which is returned to the Simulation class using the GeneratorState enumeration (Figure 3, section 3). Suppose the state of the generator (GeneratorState) returns Complete(R) (Figure 3, section 4). In that case, the generator has finished its execution and should not be resumed (failure to do so will result in the simulation being forcibly stopped). Therefore, the Simulation invokes the remove() method of Container to

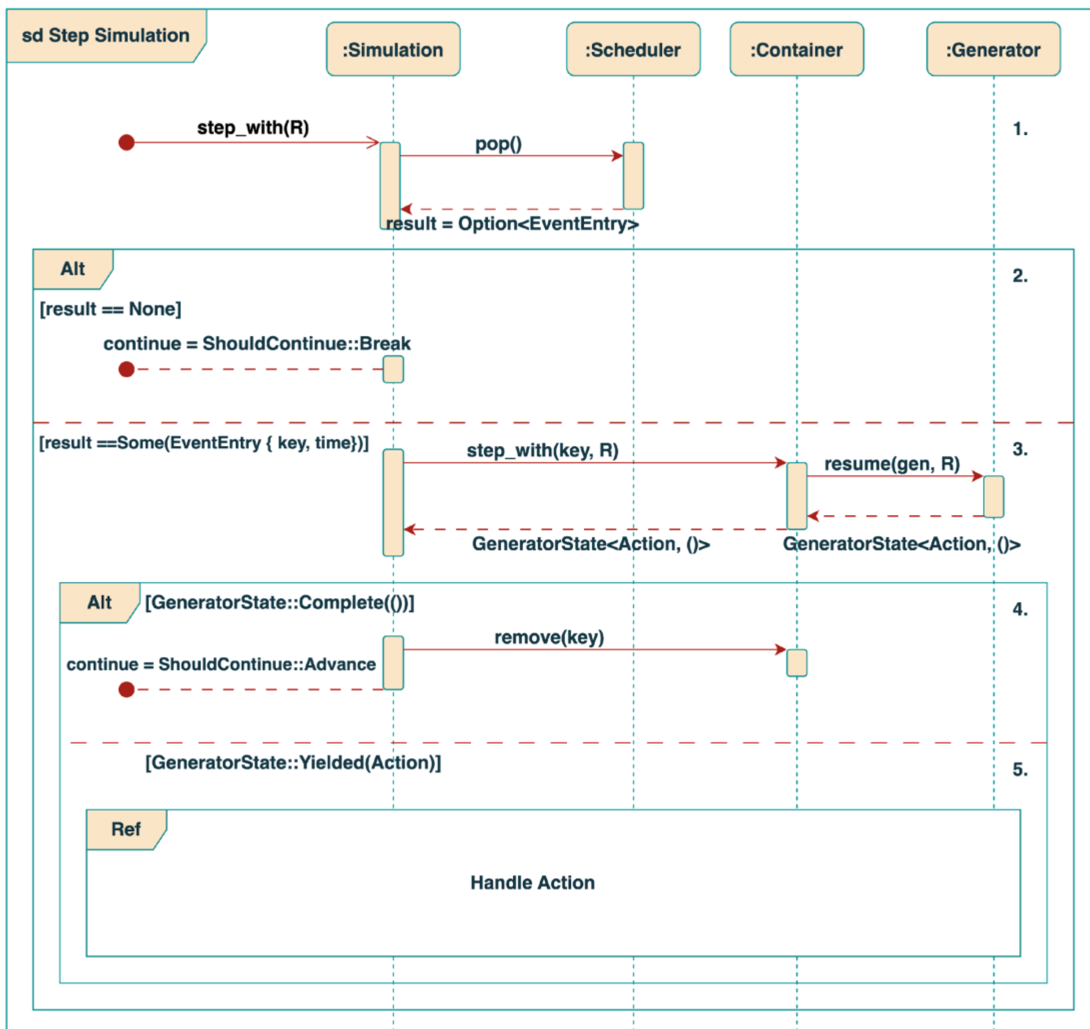

Figure 3: Sequence diagram to start a simulation in the RustSim simulation library.

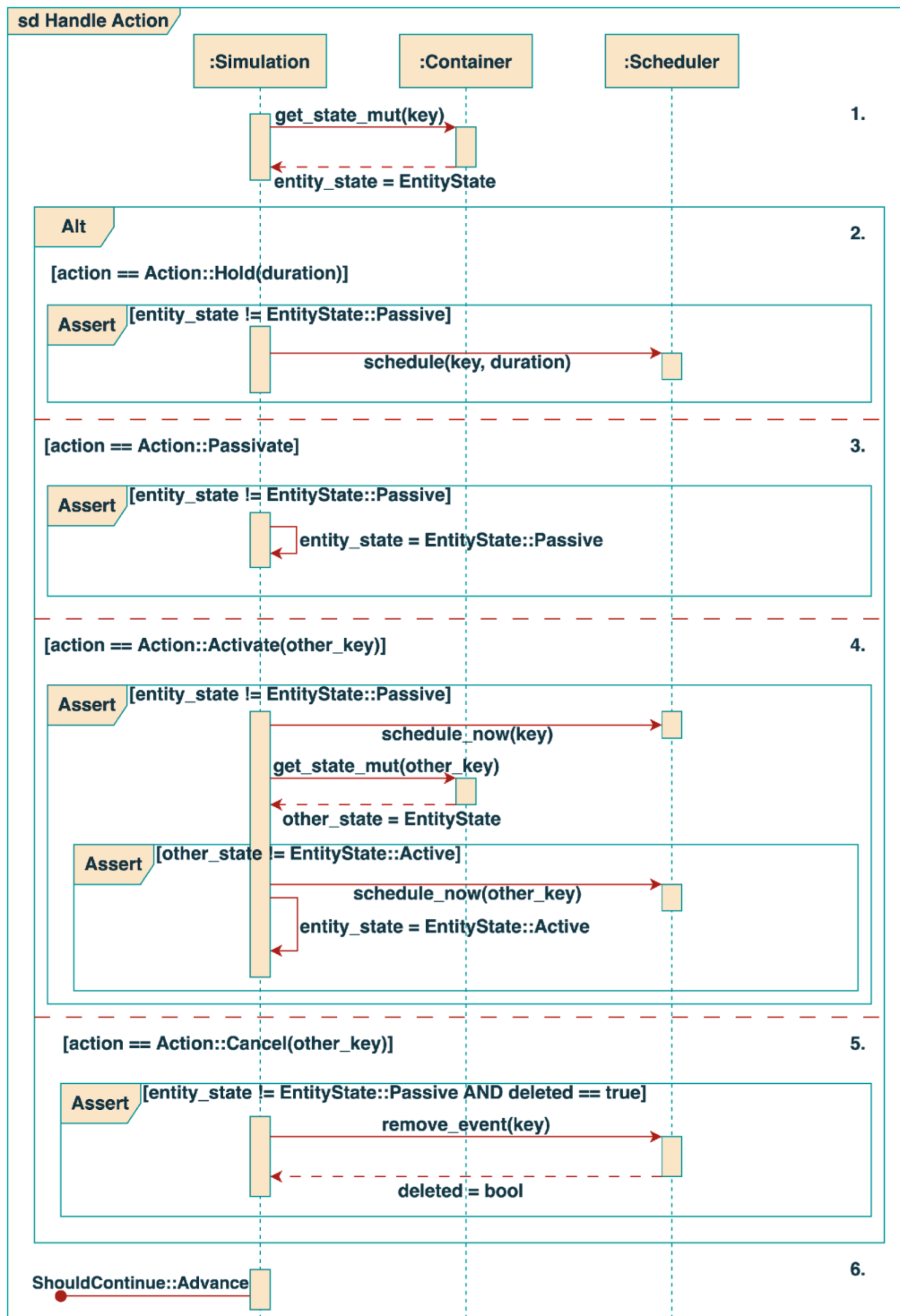

Figure 4: Sequence diagram from the internal process handling of RustSim.

delete the generator and recover the memory it was using. Finally, the Simulation class returns ShouldContinue::Advance because it has successfully processed a simulation step and can proceed to the

next one if there is any. If the state of the generator (GeneratorState) returns Yielded(Action) (Figure 3, section 5), it indicates that the generator has used yield to pause its execution. In this case, RustSim's handle action mechanism is activated (Figure 4).

We separate RustSim's handle action mechanism into six sections, where only the first and last are mandatory. All the other four sections are mutually exclusive, and only one is executed. The first mandatory step acquires the current state of the generator, which can be Active or Passive (Figure 5, Section 1). This state is used by subsequent steps to protect the simulation execution from the modification of entities by a Passive entity (i.e., a Passive entity cannot perform any actions). If the received action is an Action::Hold(duration) (Figure 4, Section 2), the schedule() method of the Scheduler is invoked, which schedules the generator at the current time of the simulation plus the duration. If the received action is Action::Passivate (Figure 4, Section 3), in this case, the generator's state is checked to ensure it is not already Passive and then changed to Passive. If the received action is Action::Activate(other_key) (Figure 4, section 4), the schedule_now() method is invoked to schedule the current generator to avoid interrupting its execution (to avoid an implicit Passivate). Then, the state of the generator associated with other_key is checked to ensure it is not Active, and schedule_now() is executed to schedule the activated generator, changing its state to Active. If the received action is Action::Cancel (Figure 4, section 5), the remove_event() method of the Scheduler is invoked, returning true if it successfully finds and removes an event associated with the provided Key or false if it finds no events. Finally, having performed an Action, the Simulation class returns ShouldContinue::Advance if it successfully processed the associated event (Figure 4, section 6).

More details regarding the implementation of RustSim can be found in our public RustSim GitHub repository.

4.3 Case Study

In this subsection, we present a POS prototype in RustSim consisting of two entities A and B. The prototype works as follows:

1 Entity B starts the simulation by doing a Passivate.
2 Entity A performs a Hold of 5 seconds and then checks if Entity B is in Passivate.
3 If B is Passivated, entity A will emit an Activate to B.
4 Independent of the condition in (2), A will do a Passivate.
5 Repeat step (2) after being Activated by B.

To construct this prototype, three elements need to be implemented: Entities A, B, and Simulation. The latter is where the simulation process (steps 1 to 5 of the list) is defined. For lack of space, only the most essential instructions for Simulation and Entity A are shown in fn main() and fn entity_a(…), respectively. This example's complete code (including comments) is available in a file called simple_model.rs under the examples directory in our public RustSim GitHub repository.

```
1. fn main(){
2.   let mut simulation: Simulation<()> = Simulation::default();
3.   let shared_state = simulation.state();
4.   let mut state = shared_state.take();
5.   let entity_b_key = state.insert(None);
   . . .
6.   let entity_states = state.insert(Passivated { entity_a: false, entity_b: false });
   . . .
7.   let a_key = simulation.add_generator(entity_a(Rc::clone(&shared_state),
     entity_b_key, entity_states));
8.   let b_key = simulation.add_generator(entity_b(Rc::clone(&shared_state), a_key,
     entity_states));
```

```
  . . .
9.   simulation.schedule_now(b_key);
10.     simulation.schedule_now(a_key);
  . . .
11.     simulation.run_with_limit(Duration::from_secs(60));
}
```

The example starts by executing the code of `fn main()`:

- The whole simulation is created in line 2.
- Since entities need to access each other, lines 3 and 4 generate and initialize a structure to store (and share) the state of entities in the simulation. The rationale for such a structure is due to Rust's restricted ownership features, which forbid direct access from one entity to another and also to avoid circular dependency among A and B.
- In line 5, space is added in the simulation shared state for the posterior allocating state of entity B (entity A will be added later).
- Another structure is declared to track whether entities A and B are Active or Passive (line 6).
- The generators that support the process approach for entities A and B are instantiated (lines 7 and 8). It is during their instantiation that the keys of each entity are passed to the other, providing access to each other.
- Entities A and B are scheduled to run at the beginning of the simulation (lines 9 and 10). Finally, in line 11, the simulation starts and is executed for 60 seconds of simulated time.

```
12.   fn entity_a(shared_state: Rc<Cell<State>>, entity_b_key: StateKey<Option<Key>>,
          entity_states_key: StateKey<Passivated>) -> GenBoxed<()> {
13.  Box::new(move |_|{
  . . .
14.  loop {
15.   yield Action::Hold(Duration::from_secs(5));
  . . .
16.   if entity_states.entity_b {
17.     yield Action::ActivateOne(entity_b_key);
18.   } else {
19.   }
        . . .
20.   yield Action::Passivate;
   . . .
21.  }
22.})
23.}
```

The behavior of entity A is described in `fn entity_a(...)`:

- The entity is declared (lines 13 and 14).
- The "infinite" processing of entity A is enabled by the loop ranging from lines 15 to 22 (depicted in Figure 5).
- In each iteration, the entity performs a hold for 5 seconds of simulated time (Line 16).
- When the time is up, it checks if entity B is passivated (line 17). If so, it activates it (line 18).
- Finally (line 21), it passivates itself until activated by B or until the simulation ends. Entity B is analogously defined.

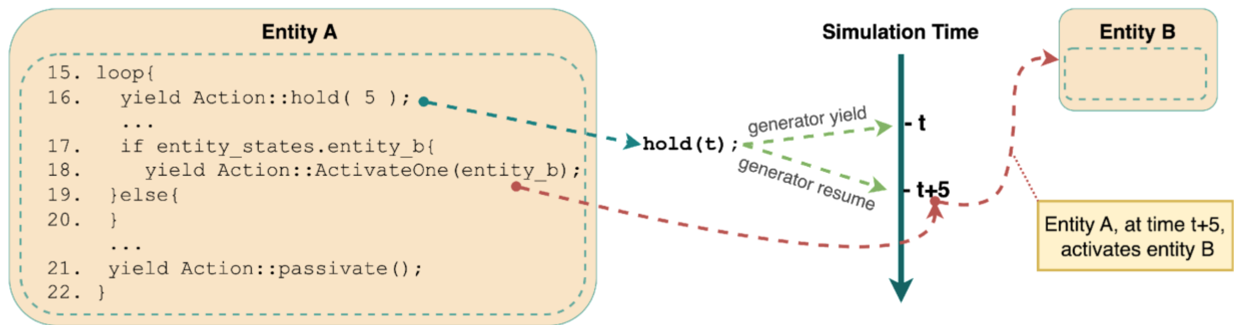

Figure 5: Generator yield and resume actions triggered by primitives called by Entity A.

5 CONCLUSION

This work proposes a new library for discrete-event process oriented simulations. The library has been designed and implemented in the Rust programming language, which has features to avoid inconsistencies of states in the entities of the models that other libraries in the ecosystem do not provide. The use of the Rust programming language provides a unique advantage over other languages due to its safety guarantees while maintaining high-performance characteristics. Memory safety and minimizing common programming problems like null pointer dereferences, buffer overflows, and data races were major design considerations for Rust. The "ownership system," which the language uses to enforce stringent ownership and borrowing laws, enables the compiler to identify potential flaws at compile-time rather than during runtime. The use of Rust allowed for efficient and safe implementation of the library, with satisfactory performance characteristics. The proposed library provides a flexible and extensible framework for building simulation models, with support for several types of events and entities. The implementation is based on the State design pattern, which allows for easy management of entity lifecycles. In addition, the library includes functions for initialization of the simulation, handling events, managing time, and managing entities and their states.

RustSim is a POS library devised for simulating problems that can be defined as models of processes and resources for various domains (like call centers, web search engines, or any queuing network problem). Regarding its limitations, it is not suitable to simulate spatially explicit models that involve movements of entities along a given path. RustSim is different from commercial software packages (e.g. Arena, Simio, Etc.) as it is only a programming library. As such, it requires a more profound programming background from the modeler but produces specific purpose simulation programs that consume fewer resources and are more efficient. The flexibility and extensibility provided by this library make it suitable for a wide range of applications such as manufacturing, logistics, healthcare, and more.

In future work, we can mention the increase in modularity of the library by allowing for greater flexibility in its internal structures. These structures are currently hardcoded and cannot be modified. For example, the Future Event List (FEL) is always a BinaryHeap, and its behavior cannot be changed. Similarly, the generator container is currently implemented as a vector, but it could be replaced with a hash map or another suitable data structure. Another improvement would be to allow users to define their types for yielding, which would be translatable to one of the existing actions (Hold, Passivate, etc.) as defined by the library. Currently, the library defines the Enum type for yielding, which limits users' ability to customize their simulation models.

Additionally, the library's support for resources could be improved, although the current implementation is highly usable. A recent update to the Desim library introduced a clever novel approach to resource support, which should be considered for integration into this library. Finally, we will perform a new set of experiments with a substantial number of instances to test the library under a stress scenario, keeping in mind that potentially the maximum number of instances is limited by the system memory.

ACKNOWLEDGMENTS

This work has been partially supported by Fondecyt de Iniciación 11230961 from ANID, Chile.

REFERENCES

Cassandras, C. G., and S. Lafortune. 2008. *Introduction to Discrete Event Systems*. 2nd ed. New York: Springer.

Dahl, O.-J., and K. Nygaard. 1966. "SIMULA: an ALGOL-Based Simulation Language". *Communications of the ACM* 9(9):671–678.

Gil-Costa, V., A. Inostrosa-Psijas, M. Marin, and E. Feuerstein. 2013. "Service Deployment Algorithms for Vertical Search Engines". In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 140–147.

Gordon, G. 1978. "The Development of the General-Purpose Simulation System (GPSS)". In *History of programming languages*, 403–426. New York: Association for Computing Machinery.

Helsgaun, K. 1999. "A Portable C++ Library for Coroutine Sequencing". Roskilde Universitetscenter, Department of Computer Science.

Klabnik, S., and C. Nichols. 2018. *The Rust Programming Language*. USA: No Starch Press.

Lackner, M. R. 1964. *Digital Simulation and System Theory*. System Development Corporation.

Liu, J. 2020. "Simulus: Easy Breezy Simulation in Python". In *2020 Winter Simulation Conference (WSC)*, 2329–2340.

Marin, M., V. Gil-Costa, A. Inostrosa-Psijas, and C. Bonacic. 2019. "Hybrid Capacity Planning Methodology for Web Search Engines". *Simulation Modelling Practice and Theory* 93:148–163.

Marzolla, M. 2013. libcppsim. https://www.moreno.marzolla.name/software/libcppsim-0.2.5.tar.gz, accessed 16th March 2023.

Marzolla, M. 2004. "libcppsim: a Simula-Like, Portable Process-Oriented Simulation library in C++". In *Proceedings of ESM 2004*, Volume 4, 222–227.

Marín, M., V. Gil-Costa, C. Bonacic, and A. Inostrosa. 2017. "Simulating Search Engines". *Computing in Science & Engineering* 19(1):62–73.

Nicol, D. M. 1993, apr. "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations". *J. ACM* 40(2):304–333.

Ovando-Leon, G., L. Veas-Castillo, M. Marin, and V. Gil-Costa. 2019. "A Simulation Tool for a Large-Scale Nosql Database". In *2019 Spring Simulation Conference (SpringSim)*, 1–12.

Rigo, A., and C. Tismer, 2021. "greenlet: Lightweight Concurrent Programming".

RustSim. Rustsim GitHub repository. https://github.com/moyarzunsil/RustSim, Accessed on April 12th, 2023.

Saydam, T. 1985, apr. "Process-Oriented Simulation Languages". *SIGSIM Simul. Dig*. 16(2):8–13.

Schwetman, H. 1986. "CSIM: A C-Based Process-Oriented Simulation Language". In *Proceedings of the 18th conference on Winter simulation*, 387–396.

Schwetman, H. 1990. "Introduction to Process-Oriented Simulation and CSIM". In *Proceedings of the 1990 Winter Simulation Conference*.

SimPy, T. 2007. SimPy: Discrete-Event Simulation for Python. http://simpy.readthedocs.org/, accessed 3rd March 2023.

SimPy, T. 2017. "SimPy: Discrete-Event Simulation for Python". Python package version 3(9):7.

Wainer, G. A. 2017. *Discrete-event modeling and simulation: a practitioner's approach*. Florida: CRC press.

AUTHOR BIOGRAPHIES

KEVIN FREZ is a senior undergraduate student of Computing and Informatics Engineering from Universidad Arturo Prat. His research interest is discrete-event simulation. He can be contacted at kfrez@estudiantesunap.cl.

MAURICIO OYARZÚN received the Ph.D. degree from Universidad de Santiago, Chile. He is an Assistant Professor at the Faculty of Engineering and Architecture at Universidad Arturo Prat, Chile. His research interests include information retrieval, index compression, and data structures for discrete-event simulation. He can be contacted at moyarzunsil@unap.cl.

ALONSO INOSTROSA-PSIJAS received the Ph.D. degree from Universidad de Santiago, Chile. He is an Associate Professor at the School of Informatics Engineering at Universidad de Valparaiso, Chile. His research interests are discrete-event and parallel/distributed simulation. He can be contacted at alonso.inostrosa@uv.cl.

FRANCISCO MORENO received the Ph.D. degree from Universidad de Santiago, Chile. He is an Associate Mathematics and Computer Science Professor at Universidad de Santiago, Chile. His research interests are information theory and multivariate statistics and their applications. He can be contacted at francisco.moreno@usach.cl.

GABRIEL WAINER received the Ph.D. degree from Université d'Aix-Marseille III. He is a Full Professor at Carleton University. His current research interests relate to modeling methodologies and tools, parallel/distributed simulation, and real-time systems. His e-mail is gwainer@sce.carleton.ca.