Seamless Data Persistence in Simulation Models: A Metaprogramming Approach in Julia

Piet Van Der Paelt[1,2][0009−0007−0902−5748], Ben Lauwens[1][0000−0003−0761−6265], and Beat Signer[2][0000−0001−9916−0837]

[1] Royal Military Academy, Renaissancelaan 30, Brussels, Belgium
piet.vanderpaelt@mil.be, ben.lauwens@mil.be
[2] Web & Information Systems Engineering Lab, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium
bsigner@vub.be

Abstract. Simulation software addresses complex problems that are difficult to solve analytically. Due to probabilistic elements, simulations often require multiple runs, generating critical decision-making data. However, many users are unfamiliar with the necessary persistence technologies to store and access this data, hindering effective utilisation. We propose a transparent data persistence architecture integrated into the ConcurrentSim package for the Julia programming language. By leveraging Julia's metaprogramming capabilities, we dynamically generate an object-relational mapping (ORM) model, automating data storage without requiring user expertise in persistence technologies. We provide two solutions for accessing the data, including a web-based interface for immediate access and a REST API for broader integration with external frameworks. The REST API allows for seamless incorporation of simulation data into diverse workflows, offering flexibility and ease of use. Our contribution improves the usability of the ConcurrentSim ecosystem and demonstrates the power of macro expansion in Julia for creating dynamic ORM configurations. Thereby, we simplify data persistence and enhance user productivity by removing most of the technical overhead, enabling users to focus on analysis.

Keywords: Discrete Event Simulation · Persistence · ConcurrentSim · Object-Relational Mapping · ResumableFunctions · Metaprogramming · Julia

1 Introduction and Motivation

The rationale behind simulation is to collect data on systems or processes that are more likely to be described by a model which can be evaluated numerically rather than being subject to exact mathematical methods which produce an analytic solution [15]. Therefore, the goal is to generate data by means of simulation, which, in turn, allows for conclusions to be drawn about the system from which the model originated.

We focus on the ConcurrentSim [12,14] Discrete Event Simulation (DES) framework implemented in the Julia programming language [3]. This framework has been inspired by SimPy [11] and DISCO [8], which is a SIMULA [5] class for combined continuous and discrete simulation. The framework has been adopted in diverse research projects situated in several domains, including the medical [6], human resources [2], ballistics [9] and network dynamics modelling[3] domains.

Abar et al. [1] presented an extensive list of Agent-based Modelling and Simulation (ABMS) tools, both open-source and proprietary, with a categorisation based on the model development effort. Furthermore, Abar et al. stated that *"An ideal simulation system should require minimal learning effort as well as provide flexible support to creating models and running robustly on any type of computing machine"* and we share that thought. Moreover, since ConcurrentSim is a process-oriented framework where a simulated entity can be seen as an independently coded agent, ConcurrentSim deserves its place amongst other ABMS tools.

We think that Abar et al.'s statement can be extended to monitoring and persistence. To the best of our knowledge, none of the open-source frameworks provides an easy-to-implement data storage mechanism. Monitoring creates an additional burden for an end user due to the required knowledge of persistence technology and how to interact with it from within the simulation model, regardless of the chosen programming language or implementation. Our goal is to lower that burden by removing this requirement. Currently, ConcurrentSim lacks the functionality to transparently store state variables characterising the evolution of a running DES model. To mitigate this shortcoming, we extended both the `ConcurrentSim.jl` [12,14] and `ResumableFunctions.jl` [13] packages by implementing a transparent probing and data persistence architecture, making use of Julia's metaprogramming functionality [18,19]. The high-level architecture of our solution is outlined in Fig. 1.

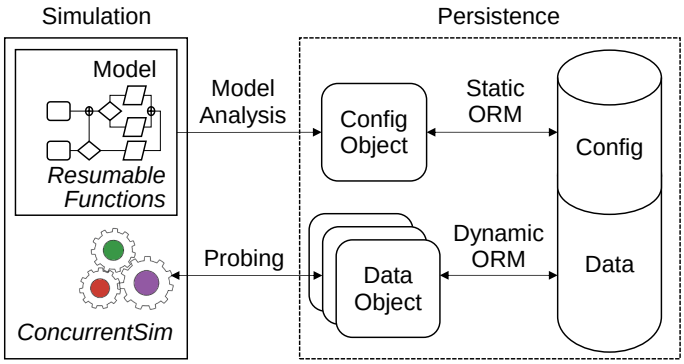

Fig. 1. High-level persistence architecture

[3] `https://qs.quantumsavory.org`

At a high level, we distinguish a static phase, during which model analysis and metadata extraction takes place via a modification of `ResumableFunctions.jl`, from a dynamic phase, where a simulated system is probed to retrieve generated data by employing an added callback function in the `ConcurrentSim.jl` package. To store the model metadata (represented by config objects) and simulation-generated data (represented by data objects), both phases make use of the Object-Relational Mapping (ORM) concept [16] offered by the `PostgresORM.jl` package[4], supported by the PostgreSQL Relational Database Management Systems (RDBMS)[5]. Transparency for end users is achieved through metaprogramming, which drives the dynamic phase by using the config data extracted in the static phase

We start by providing an overview of related work and presenting the three main packages playing a central role in our work. This is followed by the presentation of our architecture for probing and persistence in `ConcurrentSim.jl` simulations. The technical evaluation describes use cases and recorded outcomes. The subsequent discussion section explains the rationale behind some of our important decisions and choices. Finally, we conclude with the improvements acquired through our new persistence architecture.

2 Related Work

Booth et al. categorise persistent data as data that outlives the application, both in time and space [4]. Once data is generated in an object-oriented model, it must remain available in a data container for later use. In existing applications, this data container is often an RDBMS. Ireland et al. describe the *impedance mismatch problem*, particularly for the object-relational case, and mention ORM as a technique to address this problem [10]. The impedance mismatch problem refers to the challenges that arise when implementing a software artefact that relies on distinctive technologies using different data representation and storage paradigms that were not designed to work together in the first place.

An alternative to an RDBMS used as data containers were db4o and other object-oriented databases. As described by Hauser et al. [7], db4o represented another approach to solving the object-relational impedance mismatch problem. Object classes can immediately be stored in the database without any further intervention. Primarily conceived as a library for a one-tier approach, db4o's main goal was to provide applications with a small footprint data container. Despite the possibility of running in a two-tier client-server mode, db4o was not to be seen as an alternative to an RDBMS due to its inability to run in a high-available mode in a data centre or the lack of native authorisation support.

The Jakarta Persistence API (formerly Java Persistence API or JPA) provides a well-maintained alternative. Saeed et al. [17] discussed the three main aspects that make up the API. *Data modelling* concerns the annotations necessary to transform Plain Old Java Objects into entities. *Data persistence* treats

[4] `https://juliapostgresorm.github.io/PostgresORM.jl/`
[5] `https://www.postgresql.org`

how instances of these entities can be stored in the underlying RDBMS. Finally, *data querying* is about retrieval of the persisted objects using a dedicated query language. Closely related is the Hibernate ORM[6]. Despite being a native implementation, Hibernate also implements the JPA specification.

When moving towards Julia, we turn our attention to `DataFrames.jl`[7], which is the preferred method of working with tabular data in Julia, and `JuliaDB.jl`[8], a persisted tabular data package based on CSV files. The latter shares many of `DataFrames.jl`'s properties. However, these two solutions do not solve the object-relational impedance mismatch problem since they do not provide any means to persist simulation data in an RDBMS. Therefore, these two solutions are inappropriate for our purpose.

A promising alternative is `SearchLight.jl`, representing an ORM backend for the `Genie.jl` package for Model-View-Controller (MVC) web applications in Julia. Both are part of the Genie framework[9]. SearchLight disposes of several implementation libraries integrating MySQL, PostgreSQL and SQLite databases. However, in practice, SearchLight is tightly coupled to Genie. The segregation of the domain object definitions and persistence API is implemented, but each separate type needs its own module for both the object definition and persistence API.

The impedance mismatch problem between object-oriented data models implemented in Julia applications and the PostgreSQL RDBMS has been addressed by Laugier et al. in the `PostgresORM.jl` package. As stated by Russell [16], PostgresORM relies on a Julia module providing the object classes and another module providing the persistence API. Lauwens et al. have developed the `ResumableFunctions.jl` [13] and `ConcurrentSim.jl` [12,14] Julia packages on which the simulation applications we target rely on.

We believe there is a gap to be filled regarding the persistence of data generated by simulation applications. ConcurrentSim and ResumableFunctions are advanced, as proven by their adoption in other simulation research projects [6,2,9]. The choice to pursue transparent data persistence in a simulation environment provided by the Julia ecosystem appears legitimate, given that Julia solves the two-language problem. The two-language problem concerns the paradox between fast programming languages, such as C and Fortran, and high-productivity but slower languages, such as Python, MATLAB and R. Julia's main reason of existence is its capability to solve this paradox to a large extent. PostgresORM could provide a statically configured persistence interface for an RDBMS. However, in such a scenario, Abar et al.'s concerns about learning effort and model creation [1], which we extended to the monitoring and persistence aspect, remain valid. Therefore, we decided to implement a transparent architecture where the issue is mitigated through an automated probing and persistence mechanism.

[6] `https://hibernate.org`
[7] `https://dataframes.juliadata.org`
[8] `https://juliadb.juliadata.org`
[9] `https://qs.quantumsavory.org`

The novelty of our work is primarily at the level of this transparency and how it is achieved. The simulation model's metadata, extracted without any user intervention, is used to create the necessary structures to persist the simulation data. To store the live simulation data, object definitions must be foreseen and related to the tables where they will be stored. This requirement is typically fulfilled by an ORM "handcrafted" for specific needs. Our solution employs Julia's metaprogramming support to drive code generation at compile time for each ORM related to distinctive simulation models. Lastly, when the model runs, it is probed at each step through an extension of the simulation library with an automated probing mechanism. In this way, we achieved complete transparency for end users about data persistence.

In the remainder of this section, we present the three most relevant Julia packages before presenting our solution.

2.1 ResumableFunctions.jl

`ResumableFunctions.jl` contains the `@resumable` and `@yield` macros, transforming plain Julia functions into functions which can be interrupted during evaluation with the possibility to be resumed afterwards. The argument of the `@resumable` macro is a function definition, which, in the context of this work, represents a process we want to simulate. Several internal functions rewrite the argument function's body. The resumable character is realised by substituting the `@yield` macro with a return statement, followed by a label the iterator can jump to when resuming the execution. This new body constitutes the definition of a finite state machine. Using `MacroTools.jl`[10], it is integrated into a function which, when called, results in the Finite State Machine Interface (FSMI). The FSMI is subsequently assigned to a field in a callable mutable struct, which also contains the variables included in the FSMI, and a field which holds the FSMI's current state. Finally, a function definition is generated, bearing the same signature as the original argument function.

2.2 ConcurrentSim.jl

Previously known as `SimJulia.jl`, `ConcurrentSim.jl` is the event-driven simulation framework that we extended with automated persistence features. A simulated system consists of processes defined as (nested) resumable functions and a simulation environment. The latter holds the data structures necessary to run the simulation. The simulation environment is a mandatory argument for each of the resumable functions.

The top-level process is initialised through an instantiation of its corresponding callable mutable struct as an argument to the `@process` macro in the global scope. Other processes can be nested in functions as well by means of the `@process` macro in a local function scope. Having defined the simulation

[10] `https://fluxml.ai/MacroTools.jl/`

model and the environment, the simulation can be run through a call to the `run` function with the simulation environment as an argument.

When evaluating the top-level process, the `@process` macro causes an event to get scheduled on the heap of the simulation environment. This heap is a priority queue provided by the `DataStructures.jl` package[11]. The event has an array of callback functions that get appended functions to be executed once the event occurs.

Finally, the `run()` function runs the simulation in a stepwise manner. The first event in the priority queue is dequeued and its corresponding array of callback functions gets executed. All events that are scheduled on the simulation heap occur in this way. Nested functions cause events to get scheduled as well, albeit with a different priority. This approach allows for the chaining of events that cycle through the states of the different processes such that the execution of the simulation corresponds to the model which defined it.

2.3 PostgresORM.jl

`PosrgresORM.jl` provides an object-relational mapping between a Julia application and a PostgreSQL database. Data lives in an application and needs to move back and forth between the database and the application itself. Standard SQL could provide a bidirectional interface between both, but this comes with the downside that a programmer needs to master the technology.

The package provides several functions which map directly to the Create, Read, Update and Delete (CRUD) operations. All of these functions take an object as an argument. Depending on the nature of the function it was passed to, the object is persisted or used as a filter object in the context of an SQL `WHERE` clause. The object possesses uniquely identifying attributes, which translate into primary key attributes in the relation.

3 Probing and Persistence Architecture

Our architecture's main design principle is to probe a running simulation model for the values of its current state variables and to persistently store this information for later exploitation. The array of callback functions gets executed after the occurrence of each event and is therefore suitable for invoking the probing function as illustrated in Fig. 2. After an event e_i gets executed, the array of n callback functions c_j gets executed, where c_n is the probing function itself.

Since active processes switch back and forth, the probe might collect a different set of variables after the occurrence of each event. For this reason, the probe needs to be aware of the kind of active process. The latter implies a need to pass the FSMI as an argument to the probing function, resulting in the correct set of state variables. This set gives rise to an object which includes the necessary fields for every process and must be persisted in the database.

[11] `https://juliacollections.github.io/DataStructures.jl/`

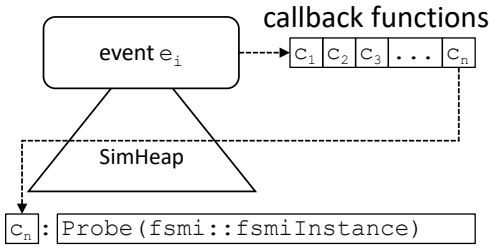

Fig. 2. Event callback functions and probing

This is closely related to the definition of an ORM, the technique of choice to persist data. Often, ORMs have a static configuration to determine which objects can exist and to which relations and attributes they map. PostgresORM applies the same approach, using modules to achieve it. In our implementation, macro expansion is applied to generate these modules dynamically at compile time. The corresponding macros take the ORM configuration as input and produce the PostgresORM-compatible modules. As such, a dynamic ORM is realised. The probing function selects a suitable object instantiator function and persists the object in the correct relation due to the mapping module. Note that multiple simulation runs can benefit from the same macro expansion.

To retrieve the necessary input driving these macros, we must analyse the corresponding model. However, it is not necessary to execute the analysis before each simulation run. If the simulation model remains unchanged, the data model and hence the state variables that need to be persisted remain stable as well.

`ResumableFunctions.jl` performs such an analysis at compile time via the `@resumable` macro. Our architecture benefits from this implementation to retrieve and store the simulation model's metadata. The model metadata is stable and needs to remain available throughout several simulation runs and must therefore be persisted separately. Thus, a classic use of a static ORM is suitable in this case.

Figure 3 shows the implementation of our novel architecture, which executes in three phases, along with the main Julia modules and functions. Each of these phases requires specific infrastructure:

1. In a first phase, the simulation model is analysed. Static programming logic and infrastructure are used to retrieve and store model metadata for internal exploitation.
2. In a second phase, the dynamic ORM is realised through macro expansion, driven by the metadata gathered in the first phase. To persist data, dynamic infrastructure, dependent on the model under consideration is created and used to persist data.
3. In a third phase, data is made externally available using the data model created in the previous phase. A REST API and a `Vue.js`-based data interface are provided.

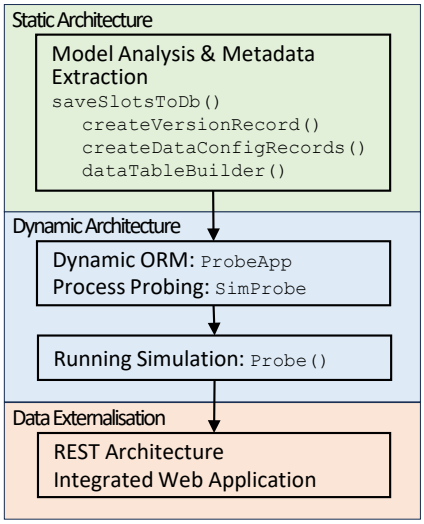

Fig. 3. Three-phased approach

In the following sections, we explain how these three phases have been realised in terms of programming logic and database configuration.

3.1 Static Architecture

The starting point for each simulation is the definition of the model as resumable functions. We benefit from this macro expansion step to store the extracted information about the state variables (slots) present in the function. For this purpose, we extended `ResumableFunctions.jl` in two ways: (1) the macro now additionally saves metadata about the model itself and the slots present in each process and (2) it takes a second boolean argument to persist or not persist the evolution on the state variables.

The `@resumable` macro internally depends on the existing `get_slots()` function. We introduced our new `saveSlotsToDb()` function, which (1) saves information about the model version through the function `createVersionRecord()`, (2) saves the configuration of these slots to the database by making use of the `createDataConfigRecords()` function and (3) creates the relations that will hold the data for the function/process under consideration through the `dataTableBuilder()` function. Figure 4 highlights the subordinate function calls of a call to `saveSlotsToDb()` and the respective database parts that are affected.

The function `createVersionRecord()` only creates a *modelmetadata* record in a dedicated relation if the model did not exist before. Otherwise, the corresponding record gets updated. The primary key for such a process (and therefore also of a *modelmetadata* record) is the function name (that was initially passed to the `@resumable` macro), along with a hash of the slots. The relation holding information about the process under consideration includes the attribute names,

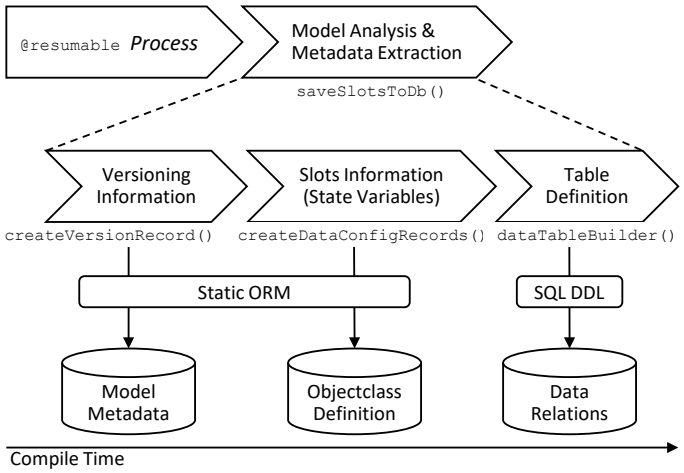

Fig. 4. Static architecture overview

a hash of the slots, a UUID, a model creation timestamp and a last used timestamp. To store the data, a static ORM was defined and is used to query the relation for the primary key of the process under consideration. Should the key exist, then the last used timestamp gets updated, otherwise a record is created.

After having saved the metadata concerning the current process, more fine-grained metadata concerning the process's specific slots is saved through the `createDataConfigRecords()` function. This function provides the necessary information to the macros that create the dynamic ORM. In the dedicated *objectclassdefinition* relation, each slot's metadata is stored, including the object and relation name, the object's field name, the data type, the mapping attribute and information whether the field forms part of a primary key. This is realised through two static ORM definitions pointing to the same relation.

Both of the relations we have considered so far are statically defined. The attributes are known upfront, which allows for a static ORM and a non-changing SQL DDL statement. The relation necessary to persist the slots of a process during the simulation is different since the attributes may vary depending on the slots present in the process. Therefore, a dynamic SQL DDL statement must be established and executed for each new process. This is the task of the `dataTableBuilder()` function, which builds and executes the SQL DDL for each process. We benefit from the SQL `IF NOT EXISTS` constraint to skip the creation of tables that already exist due to prior simulations of a certain process with the same set of slots.

The architecture requires a PostgreSQL database instance running with a dedicated schema. The schema must contain both statically defined relations, *modelmetadata* and *objectclassdefinition*. Access must be granted to a specific user for connections from static and dynamic ORM definitions. The schema holds the dynamically defined relations as well.

3.2 Dynamic Architecture

Given the availability of model and process metadata, the probing and persistence of a running simulation model is possible by using that metadata in the dynamic part of our architecture. The starting point is the `@runPersisted` macro that has been added to `ConcurrentSim.jl`, which must be used to start a simulation rather than the `run()` function. The newly added macro fulfils two functions: (1) it includes the `ProbeApp` module, which is a dynamically created module depending on the gathered metadata and (2) it designates which provided dynamic ORM object instantiator functions are used to probe which processes.

ProbeApp Module
`ProbeApp` is the module implementing the internal `Model` and `ORM` modules that make up a standard PostgresORM configuration. Figure 5 depicts the point in time when the `ProbeApp` module is loaded due to the call to the `@runPersisted` macro, which in turn results in the subsequent macro calls to `@makeObjectDefModule` and `@makeOrmDefModule`. Both of these macros are responsible for model (Object Classes) and ORM (Persistence API) creation at the same place where one would configure PostgresORM statically. These macros share the design principle that they both exploit metadata collected in the static part of the architecture described in Sect. 3.1, which alters the standard static approach of PostgresORM to the novel dynamic behaviour.

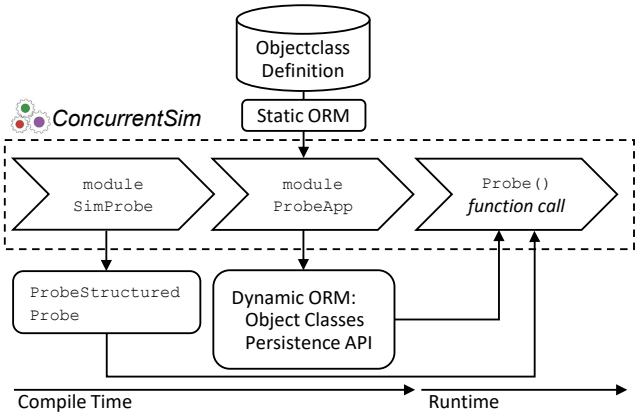

Fig. 5. Dynamic architecture overview

The makeObjectDefModule Macro
The aim of `@makeObjectDefModule` is to build a PostgresORM-compliant module containing object definitions synthesised from the records available in the

objectclassdefinition table. The task is accomplished by building an expression that evaluates to the `Model` module which configures PostgresORM. The object definitions included in this expression are created through mapping the `objToObjDef()` function over an array of metadata objects which originated from the table. Only the attributes of interest are retained in the record for metadata-object translation. The same static ORM we used previously to store data in the concerned table is now used to query it. A secondary task of the macro is to identify the current model's instantiator functions since the probe will later use only these. These are added to an array that becomes available in the global scope after evaluation of the macro.

The `objToObjDef()` function realises individual struct definitions. It does so for what is concerned with the fields of the struct using string interpolation in a quoted expression. Further, the function also uses `MacroTools.jl`'s `combinedef()` to generate both of the necessary positional and keyword constructor functions to comply with `PostgresORM.jl`'s requirements. As such, `objToObjDef()` is an aggregating function on the metadata objects.

The makeOrmDefModule Macro
The inner working of `@makeOrmDefModule` is similar to `@makeObjectDefModule`. The macro evaluates to an `ORM` module which is internal to PostgresORM and works in tandem with the object definition created through the evaluation of the `@makeObjectDefModule` discussed earlier. Each object definition requires the corresponding ORM.

Metadata objects resulting from the *objectclassdefinition* table serve as input to the `createOrmsFromDB()` function that dynamically generates the ORM modules on a per-object basis. The final module body is generated through string interpolation in a quoted expression.

SimProbe Module
Probing and persistence are realised through the `SimProbe` module, which exports the probing function and acts upon the different (sub)processes. Figure 5 shows the point in time when the probe becomes available. As illustrated earlier in Fig. 2, the probing function is activated through inclusion in the array of callback functions getting executed when an event occurs. Only after the object classes and the persistence API have become available through the inclusion of the `ProbeApp` module, as explained in Sect. 3.2, the corresponding probing function calls can be executed. This is due to the object instantiator selection mechanism, which only works when the object classes are known in the global simulation environment.

Probe Implementation
The `SimProbe` module implements the `probeStructured()` probing function with the FSMI instance as an argument. To comply with `ConcurrentSim.jl`, a `probe()` wrapper function exists as well. A mapping dictionary available at the main simulation environment struct provides information on whether the

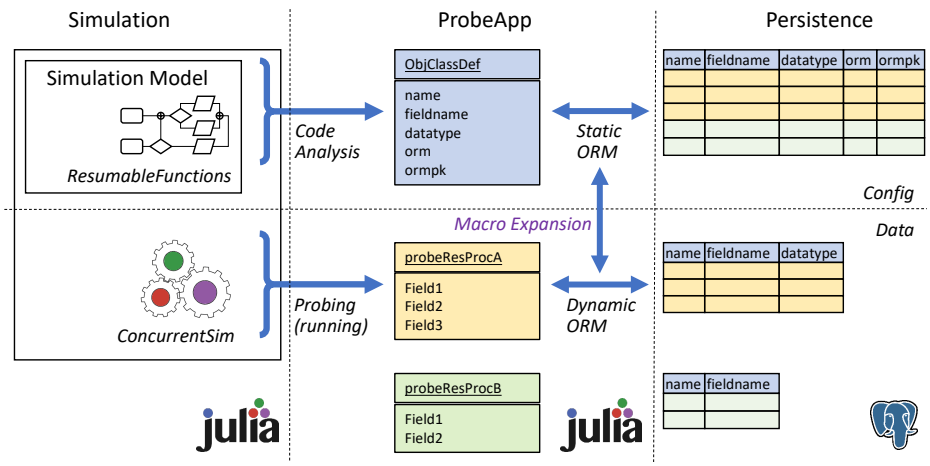

Fig. 6. Detailed persistence architecture

FSMI instance under consideration is truly a monitored process through the presence of the corresponding object instantiator function. The probing function transfers the variables present in the FSMI to a corresponding struct with the latter being persisted in the database. This mechanism relies entirely on the availability of the dynamic ORM which is available in the global scope through macro expansion from within the `ProbeApp` module, and on the coupling of FSMI types to the correct instantiators.

Probe Activation

The `probe()` function is activated using the array of callback functions. The evolution of the variables in a running simulation maps to the evolution of the state variables in the corresponding state machine. The latter represents a (sub)process contained in the simulation model. To capture state variables, the full array of callback functions must be executed first. Only then can a state machine make a transition from one state to another, or in other words, proceed to the next event.

Finally, the probing aspect and the dynamic ORM aspect are joined by coupling the type of FSMI to the correct instantiator. This is only possible when the static phase has terminated and the dynamic phase is beyond the evaluation of the `SimProbe` module. At this stage, the information on which functions must be monitored and how their ORMs should look is available. Furthermore, through the dynamic phase, the necessary ORMs are in place. The ORMs are available to the `SimProbe` module to create and store shadow objects for each monitored process. A detailed overview is provided in Fig. 6, showing the static architecture which extracts and stores model metadata through an ORM definition known upfront and then dynamically generates the ORMs based on that metadata. Object definitions and a persistence API are now available on a per-process basis.

The coupling of the type of the FSMI to the instantiator is executed just before the simulation starts. The newly added `@runPersisted` macro performs this function. It iterates over the globally defined array of processes that need to be monitored (those who "announce" themselves) and designates instantiator functions to FSMI types. This information is stored at the top-level simulation environment struct, which was extended for this purpose.

3.3 Data Externalisation

To externalise the data gathered using the techniques described earlier, we devised two distinct approaches sharing the same goal: a user should have access to all persisted data, either through an interface implementing an open standard or in tabular format from within a web browser. The first approach consists of a REST architecture, while the second approach is an integrated solution including a web application running in a single thread, mainly based on the `VueJS.jl` package.

REST Architecture
The REST architecture consists of a Julia implementation of the REST API using the `HTTP.jl` package. A HTTP server listens at a given port for a request for a virtual directory. The required table, which maps to a monitored process, is then passed to the server as a virtual directory in the requested URL and extracted using the positional pattern matching features provided by the `HTTP.jl` package. Due to the `HTTP GET` method, the server knows that the request is a READ operation on a particular table.

The base implementation is inspired by the Cross-Origin Resource Sharing (CORS) example from the `HTTP.jl` documentation[12]. However, in our implementation, a request arriving at the server invokes a `getTableContent()` service function to request the data from the back-end database. Since the `ProbeApp` module is imported in the global scope where the HTTP server runs, and the required table was passed as a positional parameter (virtual directory) in the URL, we can instantiate the correct filter object for the required table and pass it to the ORM's `retrieve_entity()` function. The resultset is then transformed to the correct JSON format using the `JSONMiddleware()` and `CorsMiddleware()` functions provided by the documentation. These two functions implement the CORS-oriented aspects as well. We refer to the documentation for the inner workings of the two latter functions.

Integrated Web Application
A router is defined in a global scope and several routes are registered on the router. Such a route combines a requested virtual directory and one of the HTTP request modes. When an HTTP client sends a request, a route invokes a function which returns a well-formatted HTML page.

[12] `https://github.com/JuliaWeb/HTTP.jl/`

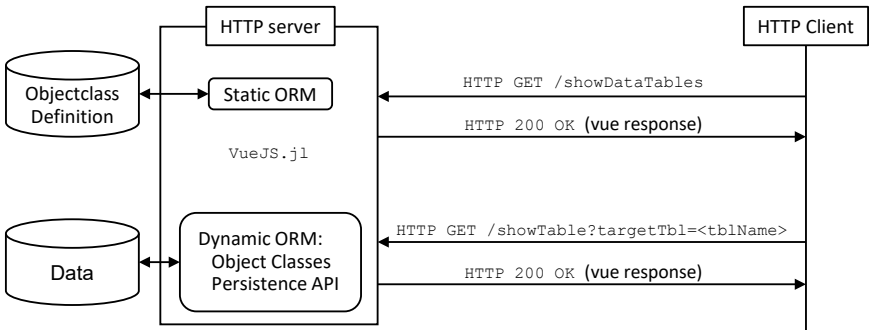

Fig. 7. Integrated web application architecture

The `VueJS.jl` package intervenes in the local scope of such a function (which is called through the request to the router), exposing several HTML elements enriched by the original `Vue.js` framework. Apart from some static definitions, our application is centred around two functions that implement the dynamic approach. Figure 7 shows the web application architecture and its intended use. The `showDataTables()` function is invoked by the `GET` request for the corresponding virtual directory `/showDataTables` and implements the landing page showing a list of all processes ever persisted in the connected schema. Figure 8 shows an example of the landing page for the simulation model used throughout this text. Upon selection of a process, a request for the `/showTable` virtual directory is sent to the HTTP server invoking the corresponding `showTable()` function, which composes the process-specific HTML page showing the requested data table.

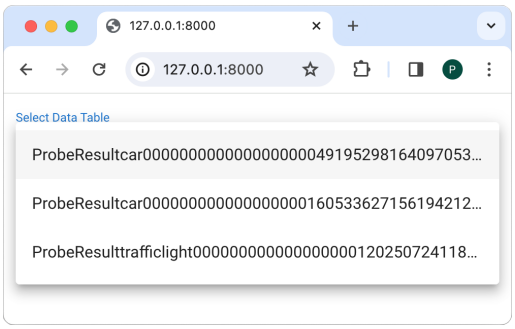

Fig. 8. Integrated web application landing page

The two functions integrate our earlier presented statically and dynamically defined ORMs. Whereas these ORMs have the role of data persistence, they are now employed in data retrieval. The `ProbeApp` module is imported into

an application's global scope. The `showDataTables()` function uses the static ORM to retrieve the list of processes available in the schema by sending a non-configured filter object to the schema. The result is used to populate the list of possible processes on the landing page. A selection of one of these processes invokes the `showTable()` function which is aware of what kind of filter object to send to the schema.

Since `VueJS.jl` is highly oriented towards dataframes, as implemented by `DataFrames.jl`, and the ORMs return a resultset comprised of objects, a utility function bridges the gap between the two. It allows sending of any filter object to a schema and receiving the appropriate data frame in return.

4 Technical Evaluation

The implementation of our architecture[13] was validated using several scenarios reflecting the intended use during simulations. We used a MacBook Pro, equipped with an Intel Core i5-5257U processor with 8GB RAM. We further used VSCode v1.84 with Julia v1.8.5 and `PostgresORM.jl` v0.5.0. The changes to `ResumableFunctions.jl` were implemented starting from version v0.6.3 and for `ConcurrentSim.jl` we used version v0.8.1. Further, the used RDBMS is PostgreSQL v13.13. Our test cases simulated a car that parks, waits for some time and then starts driving[14]. The car then encounters a traffic light, needs to stop and wait before resuming driving as illustrated in Fig. 9. The simulation is stopped after an arbitrarily chosen amount of units of time or can run indefinitely. Both the car and the traffic light are implemented as functions passed on to the `@resumable` macro.

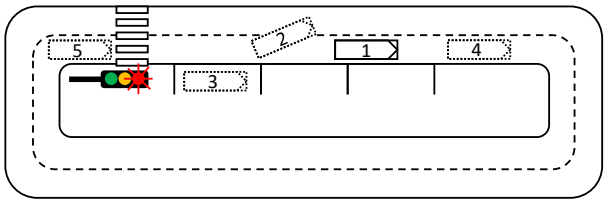

Fig. 9. Car simulation model

4.1 Inital Run

The first group of scenarios cover metadata generation and initial simulation runs. We ran the simulation model with both the `car` and `trafficlight` pro-

[13] `https://github.com/vanderpp/SimJuliaPersistence.git`

[14] `https://github.com/vanderpp/SimJuliaPersistence/blob/main/test_SimProbe.jl`

cesses unmonitored, which resulted in alterations in the modelmetadata and objectclassdefinition tables. In the modelmetadata table, records are added which indicate the same model "creationtime" and "lastused" timestamp. The object metadata to feed the dynamic ORM creation macro's is present in the second table. As expected, no data was recorded and a subsequent run of the model altered the "lastused" timestamp. We conclude that these scenarios led to the expected outcome.

Having run the scenarios directed towards metadata generation, a subsequent series of scenarios directed towards data generation was run. The preconditions for these scenarios were either none or having run the metadata scenarios. For this purpose, the `car` and `trafficlight` processes were set to "monitored", which, upon evaluation of the model, resulted in the creation and population of the corresponding data tables. All possible variations on which process was monitored during simulation were explored, with conclusive results.

4.2 Successive Runs

Starting from a model that had run previously, we devised a scenario where successive runs of the same, non-altered model were evaluated. In the metadata tables, the "lastused" timestamps were updated with every run. The corresponding data tables got appended newly generated data, distinguishable from earlier runs through the "simulation start time" attribute.

Next, an existing simulated model with persisted data was altered to evaluate the detection of the alteration and to observe the creation of new metadata records. Furthermore, we expect new data tables to accommodate the modified model data. The result for this scenario was positive. Continuing with this scenario, we reverted to the pre-existing scenario, resulting in the already existing data tables getting appended to the new data generated by the reverted model.

4.3 Data Externalisation

The starting condition for this scenario is to have a simulation that has run and data was recorded. The REST endpoint is queried using an `HTTP GET` request to the URL `http://127.0.0.1:8001/api/tablecontent/<tableName>` where `<tableName>` must be replaced with the real table name containing the data.

The integrated web application is reachable via the IP address on which HTTP server was started on. A drop-down list on the landing page was filled dynamically from the available metadata. Such a landing page example is depicted in Fig. 8, allowing for selecting any of the available processes' data.

4.4 Performance Comparison

Using the `@time` macro, we recorded the elapsed time for simulations started with the new `@runPersisted` macro. To compare performance, a series of twenty measurements were taken for each of the following configurations: non-persisted

configuration (nn) and persisted configuration (np). We also took measurements using the prior method to start simulations using the `run` function (on). All of these scenarios are based on our standard simulation.

This lead to the following mean elapsed times: $\bar{t}_{nn} = 0.0342s$, $\bar{t}_{np} = 7.2781s$ and $\bar{t}_{on} = 0.276s$. We observed a difference in means using ANOVA ($\alpha = 0.05$, $p < 0.001$). Multiple comparisons (Fisher LSD with Bonferroni) revealed a homogeneous group amongst t_{nn} and t_{on}. When used in the persisted configuration (np), our architecture is significantly ($p < 0.001$) slower than the non-persisted configuration (nn).

5 Discussion

The starting point for our architecture was the absence of a persistence architecture in `ConcurrentSim.jl` and the underlying `ResumableFunctions.jl` packages. Currently, we rely on a user's knowledge of persistence technology to implement this aspect. From our experience, we know that the presence of such knowledge is a strong assumption, which is not valid in many cases. Therefore, our goal was to integrate a fully automated and transparent solution for persistence in `ConcurrentSim.jl` and `ResumableFunctions.jl` with minimal user intervention. We identified two distinctive parts of the problem that our architecture is addressing. First, the data needs to be extracted from the running simulation and second, this data must be persisted for later use. Further, a solution should require a minimal configuration and little additional infrastructure.

For what concerns the data extraction, we must decide when the probing function should be executed. It seems obvious that the right time is the moment prior to the transition from one state to the next state. In ConcurrentSim, this is after the execution of an event which is the trigger to execute the callback functions. Therefore, ConcurrentSim's `execute()` function was altered to schedule a callback Probing function.

The probing function is exported by the `SimProbe` module, which is imported into ConcurrentSim. `SimProbe` imports the `ProbeApp` which is, in fact, the encompassing ORM module including the domain objects and the persistence API. Both are generated dynamically using macro expansion, driven by the metadata collected while evaluating the ResumableFunctions macros.

Due to the approach we took to extract metadata to drive the macros which generate the ORM modules and the requirement to hide the model analysis phase from the running simulation phase, which is probed in the background, we introduced the risk of a potential problem. This risk occurs when we want to load the module which provides the probing function with the object instantiators to persist data using the dynamic ORM. This can only be done after the model analysis has taken place. So, in the simulation application, one would need to include the `using SimProbe` statement after the model declaration and just before the `run()` function call. We decided to hide that as well using a `@runPersisted` macro to load the probing module when model metadata is available and before the probing function is scheduled as a callback.

The newly introduced architecture comes with some performance costs. Our analysis of the elapsed times for both persisted and non-persisted scenarios revealed the introduction of a non-negligible delay in the simulation process due to some I/O processes. Improvements can be made at that level by introducing some buffering mechanisms. `PostgresORM.jl` allows to flush an array of objects to the database. That approach was not explored but seems promising. Another possibility would be to run the database in memory to reduce the I/O overhead.

With regards to the decision when to consider a process as "altered" from the previous version during successive simulations, we took the approach in which the name of the function that describes the process together with a hash of the array containing the local variables of such a process determines the unicity of a process. As a consequence, when the semantic meaning of a process changes, this is not recorded in the persistence tier of our architecture. However, this can be circumvented by annotating the process using a variable. This approach is exploited to associate the proper object instantiator in the global context of the simulation application with a version of a process. When the probing function is invoked, it uses this information to decide which object to create and persist.

We investigated several approaches before integrating `PostgresORM.jl` in our architecture. The most noteworthy alternatives were `SearchLight.jl` or using a proper ORM. `SearchLight.jl` has the advantage of being an established package with a large user community. However, we believe it is too tightly coupled to Genie. The way it is implemented (using separate modules for each object and the concept of migrations to create the tables) resulted in a preference for `PostgresORM.jl`. The approach of making our own ORM was explored profoundly and led to the conclusion to cease that approach due to development time and the typical quality of code realised in such approaches. `PostgresORM.jl` gained the preference due to its straightforward approach with the domain objects module and the persistence API. We saw an ideal combination in the capabilities of Julia's macro expansions and the `MacroTools.jl` package, driven by external metadata to dynamically generate the modules at compile time.

Our implemented REST API allows processing and analysing the generated data using an external (web) application. During the implementation, we encountered the need to implement the Cross-Origin Resource Sharing solution since such an application is not guaranteed to run at the same location as the REST API. A simple REST implementation would suffice if only retrieving the JSON file would be of interest. However, our version supports both the basic and richer approach. A shortcoming of the REST API is that a user needs to know upfront the rendezvous point. A page summarising the possible rendezvous points could easily be implemented based on the available metadata.

To provide the user with an immediately available web interface to explore the stored data, we devised an architecture using Vue.js, based on the dynamically generated ORM. In this scenario, there is a landing page on which the user can select amongst the tables existing in the database, hence the simulated processes that have been persisted. The user is taken to the corresponding table, where data is available for further exploration.

A known limitation of our solution is the absence of the possibility to make certain variables non-persistent. This is a consequence of our decision to streamline the integration of the analysis and simulation phases. Should we desire to implement more fine-grained probing, the decision of what to include in the "shadow-object" used for persistence would have to be taken between these phases—after the generation of the metadata driving the dynamic ORM generation, but before the creation of the corresponding relations.

6 Conclusion and Future Work

Starting from the observed shortcomings of ConcurrentSim in its ability to easily persist simulation data and the requirement to provide such features without imposing additional knowledge requirements on the end users, we devised an architecture capable of persisting simulation data. The infrastructure and configuration requirements for using the presented architecture are minimal. The `ConcurrentSim.js` and `ResumableFunctions.js` packages were extended with modules that implement probing of a running simulation and persist it through a mechanism using a dynamically generated ORM. A running PostgreSQL instance is necessary, but the required configuration of that database schema is also minimal. As such, we were able to implement and evaluate a solution based on the presented architecture. Further, we developed two data interfaces: a REST API enabling external applications to process and explore the data and a Vue.js application supporting immediate data exploration via a web interface.

The remaining challenges for end users could be further reduced or even entirely removed by incorporating our architecture into a web application that accommodates the packages and architecture in an enclosed environment. Such an environment could facilitate model expression through a web interface. The same web application might provide easy data access after running the simulations. In that way, the remaining technical details could be hidden from end users who could focus solely on the simulation task.

References

1. Abar, S., Theodoropoulos, G.K., Lemarinier, P., O'Hare, G.M.: Agent Based Modelling and Simulation Tools: A Review of the State-of-Art Software. Computer Science Review **24** (2017). `https://doi.org/10.1016/j.cosrev.2017.03.001`
2. Abdessameud, O.M., Van Kerckhoven, J., Van Utterbeeck, F., Guerry, M.A.: Manpower Planning Using Simulation and Heuristic Optimization. In: Proceedings of ISC 2019, International Industrial Simulation Conference. Lisbon, Portugal (June 2019)
3. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A Fresh Approach to Numerical Computing. SIAM Review **59**(1) (2017). `https://doi.org/10.1137/141000671`
4. Booch, G., Maksimchuk, R.A., Engle, M.W., Young, B.J., Connallen, J., Houston, K.A.: Object-oriented Analysis and Design with Applications. SIGSOFT Software Engineering Notes **33**(5) (August 2008). `https://doi.org/10.1145/1402521.1413138`

5. Dahl, O.J., Nygaard, K.: SIMULA: An ALGOL-based Simulation Language. Communications of the ACM **9**(9) (September 1966). https://doi.org/10.1145/365813.365819
6. De Rouck, R., Debacker, M., Hubloue, I., Koghee, S., Van Utterbeeck, F., Dhondt, E.: Simedis 2.0: On the Road Toward a Comprehensive Mass Casualty Incident Medical Management Simulator. In: Proceedings of WSC 2018, Winter Simulation Conference. Gothenburg, Sweden (December 2018). https://doi.org/10.1109/WSC.2018.8632369
7. Hauser, P.: Review of db4o from db4objects. https://cis.bentley.edu/lwaguespack/CS630_Site/Downloads_files/OODBMS-db4o-Review.pdf (2011), University of Applied Sciences Rapperswil, Switzerland
8. Helsgaun, K.: DISCO-a SIMULA-based Language for Continuous Combined and Discrete Simulation: Simulation Software. Simulation **35**(1) (1980). https://doi.org/10.1177/003754978003500102
9. Hermosilla, I.: Simulating a Stochastic Differential Equation Model by Exact Sampling. In: Proceedings of MCM 2017, International Conference on Monte Carlo Methods and Applications. Montréal, Canada (July 2017)
10. Ireland, C., Bowers, D., Newton, M., Waugh, K.: A Classification of Object-Relational Impedance Mismatch. In: Proceedings of DBKDA 2009, International Conference on Advances in Databases, Knowledge, and Data Applications. Gosier, France (March 2009). https://doi.org/10.1109/DBKDA.2009.11
11. Klaus Müller, Tony Vignaux, S.S., Lünsdorf, O.: SimPy, a Process-based Discrete-Event Simulation Framework Based on Standard Python. https://simpy.readthedocs.io/en/latest/
12. Lauwens, B.: Monte Carlo Simulation Using SimJulia. In: Proceedings of MCM 2017, International Conference on Monte Carlo Methods and Applications. Montréal, Canada (July 2017)
13. Lauwens, B.: ResumableFunctions: C# Sharp Style Generators for Julia. Journal of Open Source Software **2**(18) (October 2017). https://doi.org/10.21105/joss.00400
14. Lauwens, B.: SimJulia: Discrete Event Process Oriented Simulation Framework Written in Julia. https://simjuliajl.readthedocs.io/en/stable/welcome.html (2017)
15. Law, A.M., Kelton, W.D.: Simulation Modeling and Analysis, vol. 3. McGraw-Hill New York (2007)
16. Russell, C.: Bridging the Object-Relational Divide: ORM Technologies Can Simplify Data Access, But Be Aware of the Challenges That Come With Introducing This New Layer of Abstraction. Queue **6**(3) (May 2008). https://doi.org/10.1145/1394127.1394139
17. Saeed, L., Abdallah, G.: Persistence with Jakarta EE Persistence. In: Pro Cloud Native Java EE Apps: DevOps with MicroProfile, Jakarta EE 10 APIs, and Kubernetes. Apress (2022). https://doi.org/10.1007/978-1-4842-8900-6
18. Van Der Paelt, P., Lauwens, B., Signer, B.: A Data Persistence Architecture for the SimJulia Framework. JuliaCon 2023 (Extended Abstract), Cambridge, USA (July 2023)
19. Van Der Paelt, P., Lauwens, B., Signer, B.: Model-driven Data Storage Using Dynamic Object-Relational Mapping. SIMULTECH 2023, 13th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (Abstract), Rome, Italy (July 2023)