

## Article

# Specification and Description Language Models Automatic Execution in a High-Performance Environment

Pau Fonseca i Casas <sup>1,\*</sup> , Iza Romanowska <sup>2</sup> and Joan Garcia i Subirana <sup>1</sup>

<sup>1</sup> Department of Statistics and Operations Research, Universitat Politècnica de Catalunya-BarcelonaTech, 08034 Barcelona, Spain; joan.garcia-subirana@upc.edu

<sup>2</sup> Barcelona Supercomputing Center, Plaça Eusebi Güell, 1-3, 08034 Barcelona, Spain; iromanowska@aiaa.au.dk

\* Correspondence: pau@fib.upc.edu

**Abstract:** Specification and Description Language (SDL) is a language that can represent the behavior and structure of a model completely and unambiguously. It allows the creation of frameworks that can run a model without the need to code it in a specific programming language. This automatic process simplifies the key phases of model building: validation and verification. SDLPS is a simulator that enables the definition and execution of models using SDL. In this paper, we present a new library that enables the execution of SDL models defined on SDLPS infrastructure on a HPC platform, such as a supercomputer, thus significantly speeding up simulation runtime. Moreover, we apply the SDL language to a social science use case, thus opening a new avenue for facilitating the use of HPC power to new groups of users. The tools presented here have the potential to increase the robustness of modeling software by improving the documentation, verification, and validation of the models.

**Keywords:** SDL; HPC; SDLPS; social simulation; prime numbers



**Citation:** Fonseca i Casas, P.; Romanowska, I.; Garcia i Subirana, J. Specification and Description Language Models Automatic Execution in a High-Performance Environment. *Computers* **2023**, *12*, 244. <https://doi.org/10.3390/computers12120244>

Academic Editor: Stefan Bosse

Received: 7 September 2023

Revised: 5 November 2023

Accepted: 14 November 2023

Published: 22 November 2023



**Copyright:** © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Model conceptualization is the process of describing the main causal relations between the different elements of a simulation model, using a formal or semi-formal language. Model conceptualization enables the translation of the model into code that can be executed by a computer, and also aids the validation, verification, and later accreditation phases required in any modeling project. Validation is the process of checking whether the model represents the real system accurately and faithfully, according to the objectives and assumptions of the model. Verification is the process of checking whether the model is implemented correctly and free of errors, according to the specifications and requirements of the model. Finally, accreditation is the process of certifying that the model is suitable and credible for its intended use and purpose, according to the standards and criteria of stakeholders and decision-makers.

In social simulation, the conceptualization of a simulation model, often performed in natural language (verbally), may lack the full unambiguity necessary for implementing the model in computer code. As a result, the translation from the conceptual model to computer code is prone to introduce errors. In addition, in the frame of High-Performance Computing (HPC), the code of a model must follow specific rules to enable parallelization of the code and to make use of specific libraries [1,2]. Notice that HPC is the use of supercomputers and parallel processing techniques to tackle computational problems that are too big or too hard to be solved by conventional computing methods, and the libraries and frameworks used, like OpenMP [3,4], Spark [5,6] or others can be hard. The unawareness of these libraries by context specialists who usually define the model makes it almost impossible for the automatic coding of a simulation that can be executed in a HPC environment.

This paper reports on the initial results within the frame of the European project PRACE “HPC optimization of SDLPS distributed simulator”, which aimed to develop a reliable methodology for enabling domain experts (specifically, social scientists) to model a

social system conceptually, using a modeling language called Specification and Description Language (SDL), which facilitates or automates the code generation for running the model in a High-Performance Computing (HPC) environment. We describe the main features and benefits of using SDL as a modeling language for social systems, as well as the challenges and solutions for optimizing the performance and scalability of the SDLPS distributed simulator on HPC platforms. SDLPS is a tool, developed by the Universitat Politècnica de Catalunya to execute in parallel or distributed environments for SDL models. We also present some case studies and experiments that demonstrate the applicability and effectiveness of our approach.

Some attempts have been made previously to use a high-level language to simplify the interaction with a HPC environment, e.g., Castañé et al. in [7] proposed an ontology to simplify the resource management in HPC, a similar approach to the one proposed by Faheem et al. in [8]. Employing a formal language for modeling is another solution, e.g., [9] presented an approach using Petri nets [10], again to represent the performance of the system. These approaches, however, are focused on the use of HPC platform's resources, and not on the definition of the simulation models executed on the platform. Liao et al. [11] present an ontology for HPC that aims to make training datasets and AI models FAIR (Findable, Accessible, Interoperable, and Reusable). The ontology provides controlled vocabularies, explicit semantics, and formal knowledge representations for HPC concepts and entities.

In that sense, our approach is novel since it tries to simplify the interaction between the HPC environment and the modeler rather than optimize the infrastructure. We opted for using a formal language—Specification and Description Language (SDL)—instead of an ontology, since the type of models that need HPC resources can be highly diverse, but also because SDL is particularly well versed in the Internet of Things (IoT) systems although it can be applied virtually to model any systems [12–15].

Social simulation is a broad field that covers various social contexts, such as psychology, political science, business, economics, etc. Different social simulation models have different assumptions and specific purposes of investigation. However, most of these models share some common features, such as the use of agents, rules, interactions, and outcomes. A modeling language like SDL (Specification and Description Language), proposed in this study, can be applied to any social context for any investigation purpose by allowing the user to define these features in a flexible and modular way. SDL is a graphical language that has been widely used for designing and verifying complex systems, such as telecommunications, embedded systems, and distributed systems [16]. SDL can also be used to describe social simulation models clearly and concisely, using diagrams and symbols that represent the structure and behavior of the system. The SDLPS tool also provides a library of ready-to-use procedures that can capture some common stochastic behaviors in various social science contexts, such as random sampling, probability distributions, network formation, etc. To demonstrate the usefulness of the SDL, this study uses two cases: a numerical simulation (Sieve of Eratosthenes Model) and an agent-based model (Artificial Anasazi model). The Sieve of Eratosthenes Model is a simple algorithm for finding all prime numbers up to a given limit [17]. The Artificial Anasazi model is a computational model that simulates the population dynamics and land use patterns of the Ancestral Pueblo People who inhabited the Long House Valley in Arizona from AD 800 to 1350 [18]. We do not claim that these two cases can be generalized to any case, but they illustrate how SDL as a tool can be used to model different types of phenomena that range from computational problems to social phenomena with different levels of complexity and uncertainty, and, more interestingly, in a HPC environment.

In this paper, we explain how the assumptions of each model are represented in the SDL diagrams in a graphical, complete, and unambiguous way. In particular, we will show how SDL can handle the logic and functionality of simulation models realistically and robustly, using graphical symbols, state machines, events, signals, timers, variables, data types, etc., and how we can generate code to be executed in a HPC environment. Other

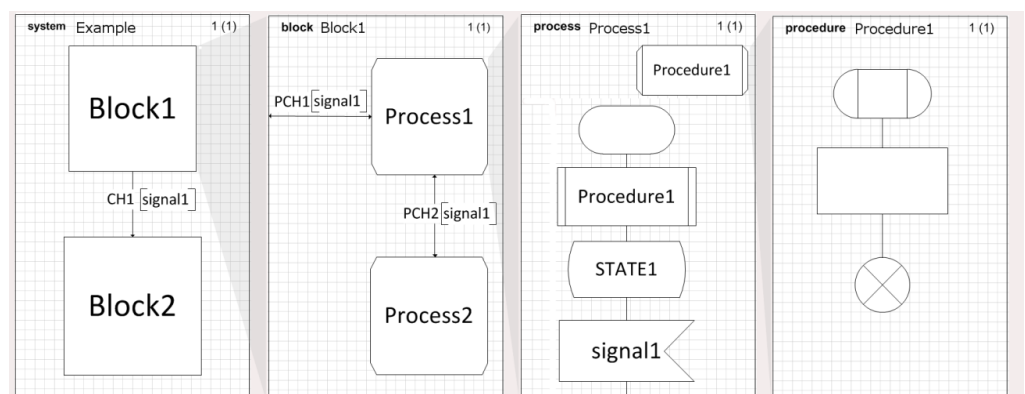
conceptualization languages exist to represent simulation models, and novel approaches are proposed to be able to optimize the code that can be executed in HPC using Discrete Event System Specification (DEVS) [19–21] or Petri nets [22–25]; however, in this paper, we are focused on how conceptualization language can be used to automatically represent models that can be executed in HPC, highlighting the advantages and limitations of this approach. Finally, we show how we can generate code from this graphical, complete, and unambiguous representation of the models that can be executed in a HPC environment, making the validation and verification processes of the model simpler and faster.

The paper is organized as follows: Section 2 presents Specification and Description Language; Section 2.1 shows the tool we used to implement the simulation SDLPS, which understands SDL; Section 3 details how we can generate code for a HPC environment with the tool; Section 4 presents the examples with the models and a discussion regarding its implementation; finally, Sections 5 and 6 contain the discussion and conclusions of this paper.

## 2. Our Approach: Specification and Description Language

Specification and Description Language (SDL) is an object-oriented formal and graphical language defined by the International Telecommunications Union–Telecommunications Standardization Sector (ITU-T) (the Comité Consultatif International Telegraphique et Telephonique [CCITT]) in the Recommendation Z.100. From its origins, SDL was designed for the specification of event-oriented, real-time, and interactive complex systems. These systems might involve different concurrent activities that use signals to perform communication [26].

Structurally, the SDL consists of four elements: system, blocks, processes, and procedures. In SDL, *blocks* and *processes* are called *agents* (Henceforth, all SDL syntax will be marked with CAPS notation. For example, when referring to a process as a dynamical element we will write it as PROCESS when representing the SDL syntax. The names for SDL model elements will be noted in *italic*). The outermost block, the *system* block, is an agent itself. Figure 1 shows this hierarchy of levels. SDL handles concurrency by allowing the user to define a system as a set of interconnected abstract machines, which are extensions of finite state machines. Each abstract machine or PROCESS has its STATES and can communicate with other PROCESSES through SIGNALS that are sent and received via gates. The SIGNALS are transmitted through CHANNELS, which can have different properties, such as delay, priority, or loss. The communication between PROCESSES can be either synchronous or asynchronous, depending on the delay of the channel. SDL also supports the dynamic creation and deletion of PROCESSES, using the CREATE and STOP primitives. SDL ensures the synchronization and coordination of the concurrent processes, using a set of rules and semantics that are precisely defined by the ITU-T standard [16,27].











**Figure 1.** A structural vision of an SDL model with four nested levels of model description [28].

Although a textual SDL representation is possible (SDL/PR), this paper uses the graphical representation of the language (named SDL/GR). More details about the Specification and Description Language can be found in Recommendation Z.100 [1] or on the website [2]. BLOCKS, PROCESS, and PROCEDURES define the basic structure and behavior of any simulation model in SDL.

The PROCESS diagram is used to represent the behavior of the model and its full ontology, as defined by the modeler. The main elements to describe this behavior are detailed in Table 1.

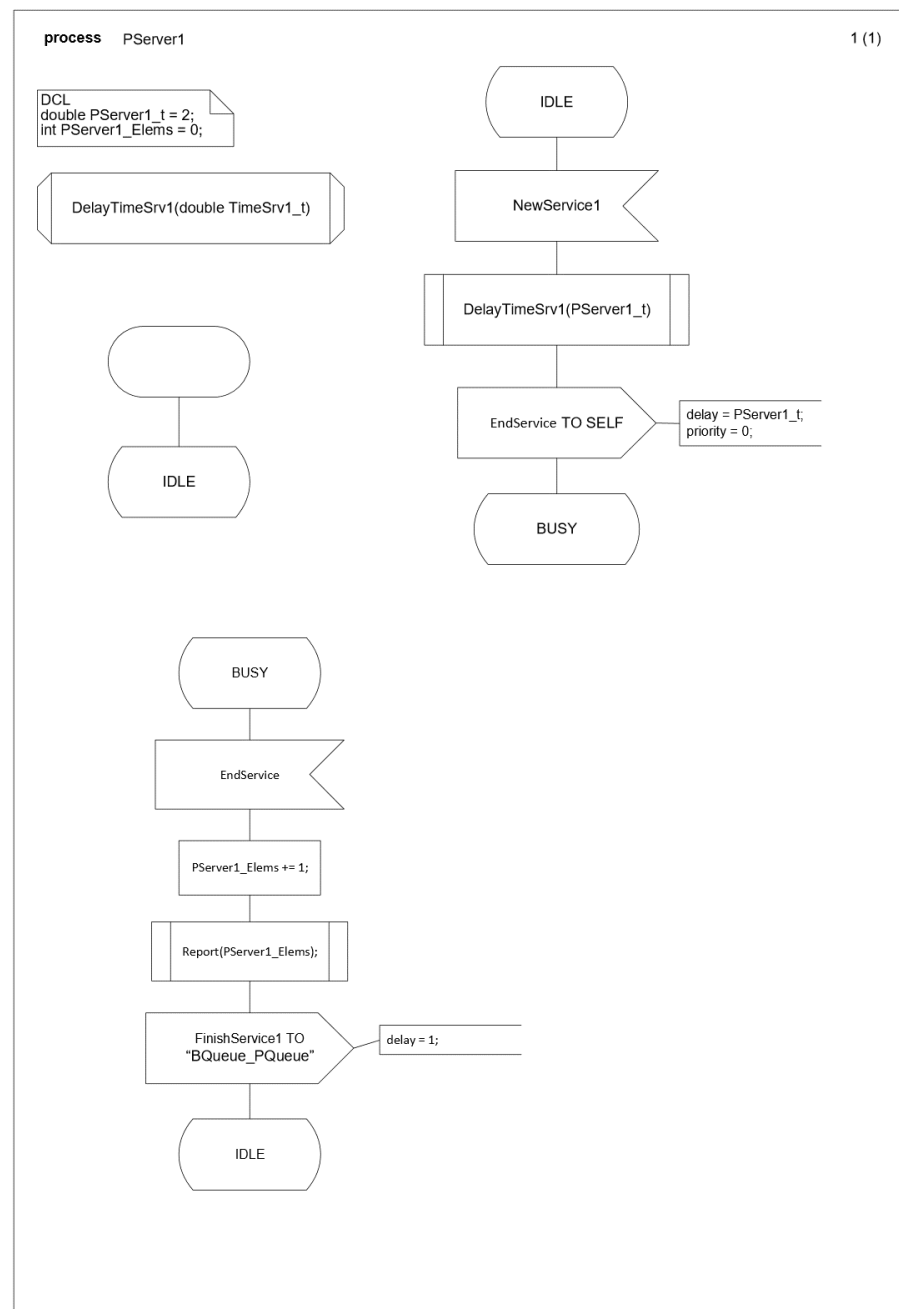
**Table 1.** Main SDL PROCESS elements.

Name	Symbol	Description
Start		This element allows defining the initial condition for a PROCESS diagram.
State		The <i>state</i> element contains the name of a state. This element defines the states of behavioral diagrams (like PROCESS diagrams).
Input		<i>Input</i> elements describe the kind of events that can be received by the process. All branches of a specific <i>state</i> start with an <i>input</i> element since an object changes its state only when a new event is received.
Create		This element allows the creation of an agent.
Task		This element enables the interpretation of informal texts or programming code. In this paper, following SDL2010, we use C code.
Procedure call		These elements perform a procedure call. A PROCEDURE can be defined in the last level of the SDL language. It can be used to encapsulate pieces of the model for its reuse.
Output		Output elements describe the kind of signals to be sent, the parameters that the signal carries, and the destination. If ambiguity about the signal destination exists, communication can be directed, specifying destinations using a processing identity value (Pid), an agent name, or using the sentence via <i>path</i> . If there is more than one path, and no specific output is defined, an arbitrary one is used. The destination value can be stored in a variable for later use. Four Pid expressions can be used: <ul style="list-style-type: none"> <li>• <b>self</b>: an agent's own identity;</li> <li>• <b>parent</b>: the agent that created the agent (Null for initial agents).</li> <li>• <b>offspring</b>: the most recent agent created by the agent.</li> <li>• <b>the sender</b>: the agent that sends the last signal input (null before any signal is received).</li> </ul>
Decision		These elements describe bifurcations. Their behavior depends on the answer to the related question.

Notice that SDL owns more elements that can be used to detail a PROCESS diagram. However, the elements presented in Table 1 are sufficient to represent any system; this can be proved because we can establish a transformation between any model represented in DEVS formalism and SDL [29]. Since DEVS is a complete formalism [30,31], this implies that SDL can also capture the essential features of any system.

Figure 2 shows an example of a process diagram.





**Figure 2.** SDL process diagram for the *ready* state of the process *PServer1*, see [32] for more examples.

The PROCEDURES diagram is the last level of an SDL formalization, allowing a description of the procedures used in the procedure calls. It is needed to perform a complete formalization of the simulation model, but usually does not add any important details regarding the main model element's behavior. Similarly to the PROCESS diagrams, it can be described with C++ code.

### 2.1. SDLPS

SDL provides a conceptual model, whose execution can be performed using tools such as PragmaDEV Studio [33,34], Cinderella [35,36] or Rational [37], among others. Here, we use SDLPS [38], a platform developed in C++ and C at the Polytechnic University of Catalonia. The model code blocks (in C for the tasks and procedures inside the SDL blocks) are used through a DLL (Dynamic Link Library). The XML code is generated from the

SDL with a plug-in on Microsoft Visio®. This means that the model can be validated and verified through graphic diagrams.

There are two crucial advantages to performing model conceptualization with SDL in cases of interdisciplinary work between context specialists and computer scientists: communication and HPC integration. First, the graphic representation of the model simplifies the interaction and communication between the different specialists ensuring that all of them fully understand the model and its functioning (Figure 3).

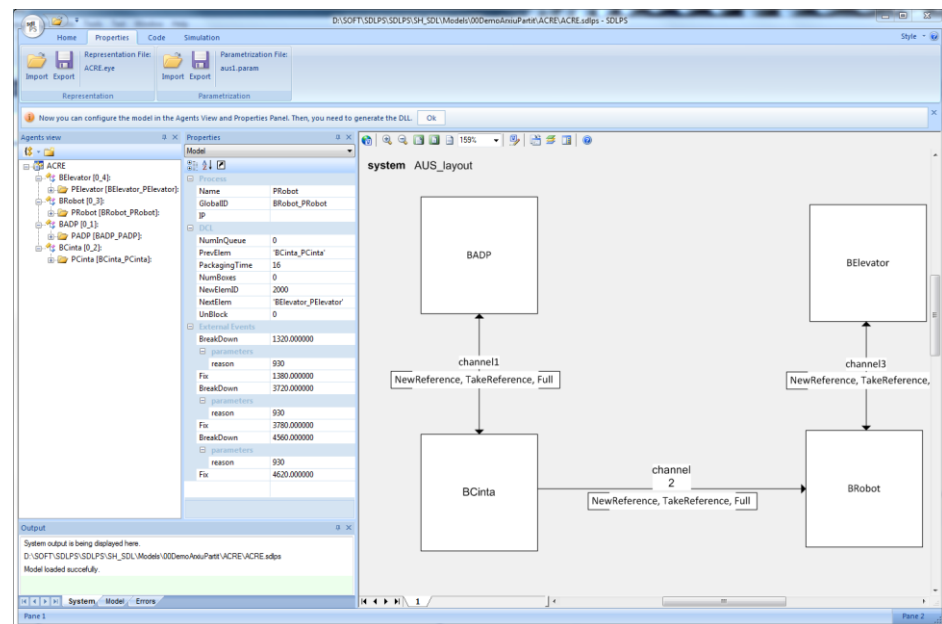
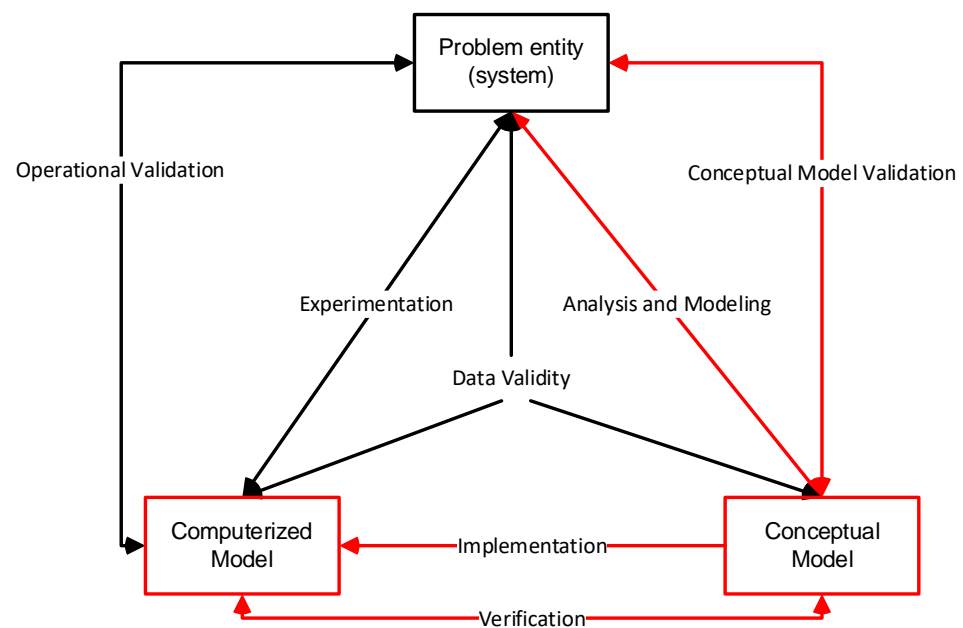


Figure 3. The SDLPS interface.

Second, there is no additional step necessary to implement the conceptual model defined in SDL in a HPC environment, since the model is automatically translated into code. This also simplifies the Implementation and Validation and Verification stages of a simulation project, (Figure 4 parts in red). Sargent's simplified Validation and Verification loop for simulation models [39] is a graphical representation of the main steps involved in ensuring the quality and credibility of a simulation model. It consists of three main elements, (i) the System, (ii) the Conceptual Model, and (iii) the Computerized Model. Several steps must be performed to guarantee the quality of the process: (i) analysis and modeling; (ii) implementation; and (iii) experimentation. In parallel, validation and verification processes must be conducted that encompasses: (i) Conceptual Model Validation—checking that the conceptual model accurately captures the system of interest and the objectives of the study; (ii) Verification—checking that the computerized model correctly implements the conceptual model and that there are no errors or bugs in the code; (iii) Operational Validation—checking that the model is suitable and useful for the intended purpose and application; and (iv) Data Validation—assuring that the data is correct for its use in the model. The simplified Validation and Verification loop is a useful tool for guiding and documenting the Verification and Validation process of simulation models. It also helps to communicate the model's quality and credibility to the stakeholders and users of the model. There are other proposals more complete [40–42], but in our approach, this is enough to express the advantages of the use of a formal language like SDL.



**Figure 4.** A simplified version of the modeling process, based on [39].

The model validation is performed with the SDL representation of the model, such that all the stakeholders involved in the project can participate in the validation process. Syntactic verification is assured because the tool recognizes the SDL diagrams.

The obtained results from the model emulation trace can be represented in any data analysis software or in the bespoke tool, SDLPSEye, which can represent information in a 3D environment. Notice that SDLPSEye only serves as a 3D representation tool, and works independent of the SDPS engine, and therefore has no effects on the calculus of the model.

#### 2.1.1.1. SDLPS Time Management

Each one of the different agents in SDL may or may not be executed in parallel, depending on its function. SDL agents that must be executed sequentially are defined as a set of PROCESSES inside of the PROCESS agent. This scheduling ensures that each agent waits for the completion of the execution of all previous agents. On the other hand, all the elements that are included within one BLOCK SDL AGENT can be executed in parallel. This implies that the AGENTS inside a BLOCK AGENT do not share memory since we do not know their order of execution.

Thus, the modeler can clearly express, graphically, the execution mode (parallel or sequential) for each one of the different simulation sub-models (SDL agents). This significantly facilitates the definition of distributed simulation models and the merging of existing simulation models with other simulation components that can actuate in real-time.

**The SDLPS clock.** SDL has a global clock that represents the simulation time. In SDLPS, this time is implemented and interpreted as the Lower Bound Time Stamp (LBTS in HLA1.3). This is the time that all AGENTS can advance securely without the need to implement a rollback mechanism. The value of this clock is stored in the SYSTEM agent to simplify the access by all AGENTS. This value exists in all the other agents that comprise the model. For all the other agents and components, this time represents its Local Virtual Time (LVT).

In our current implementation, SDLPS only supports conservative synchronization between all the elements that compose the system. However, for the model definition, it is not necessary to understand the internal time management.

### 2.1.2. Input and Output Variables

To define the communication mechanism between the different elements that compose the system, the SDL has two mechanisms depending on the nature of the sub-models. If the sub-model is defined following SDL language, we have a complete definition of its behavior and can use the PROCESS mechanism. In that case, we can use parameters that are attached to SIGNALS. When a PROCESS receives a new SIGNAL, its attached parameters can be used inside the PROCESS methods. One needs to specify what are the parameters, and the type of parameters that the different PROCESSES expect to receive.

However, often we do not have a formal representation of the model using SDL. In that case, it is needed to use a PROCEDURE CALL that connects the model with other components. This is a compelling approach as it enables the merging of external components and co-simulation mechanisms. In the SDLPS implementation of the SDL, PROCEDURE CALL can use other computer programs (or simulators) to obtain data and combine its information dynamically with the simulation model. The critical element here is that the procedures always belong to a PROCESS that defines its integration and interaction with the whole simulation model, defining the SIGNALS the parameters used in sub-models. Three different mechanisms to define the PROCEDURE are implemented on SDLPS: *full definition*, *API implementation*, and *remote implementation*.

**Full definition** means that the PROCEDURE is defined in the simulation model. As an example, a PROCEDURE may be defined as follows:

```
<!--Procedures definition.-->
<procedures>
  <procedure id="1" name="DelayTimeSrv1" implementation="">
    <params>
      <param name="TimeSrv1_t" type="double" defvalue="" ref="yes"></param>
    </params>
    <body>
      <task id="1" name="">TimeSrv1_t=60;</task>
    </body>
  </procedure>
</procedures>
```

The PROCEDURE named *DelayTimeSrv1* has a body but no specific *implementation*. This implies that SDLPS uses its body to perform actions defined elsewhere. Thus, with the full implementation, the SDLPS PROCEDURES contain all the code that is needed to be executed in the context of the simulation model defined inside the SDL diagrams (the model).

**API implementation** means that the PROCEDURE is implemented in the SDLPS itself. This approach was used for industrial [38] and environmental applications. For example, a model representing the slab avalanche phenomenon, the *GetCurCell* PROCEDURE, allows access to information of the Cellular Automaton structure used to model the avalanche, and is implemented directly in SDLSP to simplify the model definition and readability [43]. This way of modifying the model is not limited to the team of SDLPS developers. The generated DLL can include all the procedures of a model.

The last mechanism is the **remote implementation**. In this case, the PROCEDURE is implemented in another system (i.e., program). This implementation is mainly performed with the help of C++ libraries, which allow one to call the other system and retrieve their results. This program can have (or not) a connection mechanism to obtain the information. We can connect our simulator with a third-party simulator using its API, but if no API exists, other mechanisms can be implemented. As an example, we connected SDLPS with the energyPlus [44] simulation system, using the OS API instructions to access the program through an SDLPS plug-in that represents the connection point of the simulator.

Therefore, the remote implementation becomes a mechanism to glue different pieces of software. This can be similar to what we can do with the Run-Time Infrastructure (RTI) in the HLA standard. The RTI, or Run-Time Infrastructure, is a middleware that provides a standardized set of services for distributed simulation using the HLA, or High-Level Architecture standard. The RTI enables different simulation systems, called federates, to communicate and coordinate their data exchange and synchronization during a runtime execution. The RTI also supports the management and inspection of the state of a federation, which is a collection of federates that share a common Federation Object Model (FOM). The FOM defines the data types and structures that are used for information exchange among federates. To know more regarding HLA, see [45]. In the remote implementation, the SDLPS ensures the synchronization during the execution of the model, and data sharing using SIGNALS and structures defined in the declarations (DCL) SDL text area. This means that the PROCESS in the model can communicate and coordinate their actions and STATES, and access the common data and variables, according to the SDL specifications. The SDLPS also handles the parallelization and distribution of the processes and the data across the final platform that supports the execution.

In our approach, there is no need to modify the model definition, only to implement a very simple plug-in following the SDL requirements to connect the simulator (that can be a legacy infrastructure without any conceptualization) with the new simulator. In all these cases, the call to the PROCEDURE is the same. As an example, we have a PROCEDURE call named DelayTimeSrv1 that represents the time needed for a server to perform its operations:

```
<procedurecall id="2" name="DelayTimeSrv1">
  <param name="TimeSrv1_t" value="PServer1_t"></param>
</procedurecall>
```

In this code we are using a parameter named *TimeSrv1\_t* is defined in the process. Since this PROCEDURE modifies this parameter, all SDL model agents must be aware of this modification. It is indifferent to the model whether the call is synchronous or asynchronous, or if the procedure is implemented using a full definition, API, or Remote implementation. The structure of the SDL assures the coherence of the obtained results.

**SDLPS model structure.** The models defined on SDLs are structured in several directories that contain all the needed elements to execute a model.

*Doc directory.* It contains the files with the model, based on Microsoft Visio® vsdx format. These files are understood by SDLPS and used to generate an XML representation of the SDL model. This directory is the main directory of an SDLPS model since it contains the SDL/GR model.

*Code directory (SDLCodeExtenal).* It contains the files with code in C++ needed for automatic code generation, but not the model code, for example, the methods needed to access input data.

*Data directory.* It contains the files with the model input data.

*Model directory.* It contains XML files with the model generated from the SDL/GR diagrams. This directory is autogenerated.

*Traces directory.* It contains the files with the traces (raw output) of the model execution necessary to do Operational Validation.

*Root directory.* The root directory also contains an automatically generated set of files later used in the model execution, like the XMLK representation of the SYSTEM diagram (with the name of the model and "sdlps" extension) or the DLL needed for the execution of the model in a local computer.



### 3. Generating the HPC Code

The tools needed to generate the C++ code that can be executed in the HPC environment consist of a model compiler and a simulator. The former takes care of reading the XML generated by SDLPS, which represents the model to simulate and run several stages to ensure the model is correct. The latter is the infrastructure that will manage the runtime execution of the model, allowing it to run certain parts of the simulation in parallel while keeping the results the same as if it were executed sequentially.

#### 3.1. The Marenosttrum

In our specific case, the HPC is the Marenosttrum 4 Supercomputer [46] with a computing power of up to 11.15 Petaflops, and achieved 48 racks with 3.456 nodes. It is worth noting that only single-node experiments using two Intel Xeon Platinum 8160 with 24 cores each at 2.1 GHz frequency are presented in this work. Figure 5 shows the Marenosttrum 4.



**Figure 5.** Marenosttrum 4, source: Martidaniel, CC BY-SA 4.0 <<https://creativecommons.org/licenses/by-sa/4.0/>>, via Wikimedia Commons accessed on 15 November 2023.

#### 3.2. System Architecture

The model conceptualization is performed in SDL language. We used it to represent, graphically, the SDL diagrams via Microsoft Visio®, and we stored them using the standard vsdx format. The SDLPS tool opens this model, written in vsdx, and performs a transformation of the code to an XML code that represents the SDL model. Then, it analyzes this XML representation doing a first syntactic validation of the model. From this XML representation of the model (the .sdmps file), SDLPS generates C code to represent the PROCEDURE elements of the model and connect the different model AGENTS (PROCESS and BLOCKS). This code, however, is intended to be used alongside SDLPS, which is not prepared to be executed on the Marenosttrum environment. Instead of using this code, a new library allows the generation of a C++ code that is suitable for its execution in the Marenosttrum 4 Supercomputer. Detailed below is the process of how these libraries work.

First, we start from the XML representation of the SDL model, the XML-SDL model. The next lines show an XML sample of an SDL block.

```

<sdlblock>
  <block id="0" name="Bprime" IP="" portRead="" implementation="" inherits="">
    <channels>
      <channel name="" start="PManager" end="PSegment" dual="yes">
        <event name="SEGMENT"/>
      </channel>
    </channels>
    <process id="1" name="PSegment" implementation="model\PSegment.sdlps" IP=""
portRead="">
    </process>
    <process id="2" name="PManager" implementation="model\PManager.sdlps" IP=""
portRead="">
    </process>
    <DCLS/>
    <procedures/>
    <start/>
  </block>
</sdlblock>

```

The next lines show an example of an SDL process with the definition of one st

```

<sdlprocess>
  <process id="3" name="PSegment" IP="" portRead="" implementation="" inherits="">
  <DCLS>
    <DCL name="i" type="int" value=""/>
    <DCL name="L" type="int" value=""/>
    ...
  </DCLS>
  <procedures>
    <procedure id="1" name="ReportList1" implementation="">
      <params>
        <param name="list" type="int*" defvalue="" ref="yes"/>
        ...
      </params>
      <body>
        <task id="1" name=""> /* Run through each of the numbers in list1 */ for (c =
2; c <= sqrtN; c++) { /* If the number is unmarked */ if (list1[c] == 0) { /* The
number is prime, print it */ Report("%lu ", c); } } </task>
        <return id="2" name=""/>
      </body>
    </procedure>
    <procedure id="2" name="ReportList2" implementation="">
      <params>
        <param name="list" type="int*" defvalue="" ref="yes"/>
        ...
      </params>
      <body>
        <task id="1" name=""> /* Run through each of the numbers in list2 */ for (c
= L; c <= H; c++) { /* If the number is unmarked */ if (list2[c-L] == 0) { /* The
number is prime, print it */ Report("%lu ", c); } } </task>
        <return id="2" name=""/>
      </body>
    </procedure>
  </procedures>
</sdlprocess>

```

```

        </procedure>
    </procedures>
    <start>
        <task id="1" name="">sqrtN=sqrt(N);</task>
        <setstate id="2" name='PREPARING'>/>
    </start>
    <state name='CALCULATING'>
        <input id="1" name='CALCULUS'>/>
        ...
        <setstate id="6" name='RESULTS'>/>
        <output id="7" name='PARTIAL_RESULTS' self="" to="" Bprime_PSegment [0] '' via="">
            <userparam name='list2' value='list2'>/>
        </output>
        <setstate id="8" name='END'>/>
    </state>
</state name='PREPARING'>

```

SDL owns two possible representations, the graphical one, GR-SDL, and a textual one, TR-SDL. Both are equivalent. Notice that the XML-SDL representation used on SDLPS is only a representation based on XML of the TR-SDL, the textual representation of SDL.

From the XML code defined on SDLPS, the system is capable of generating the C++ code that will be executed on the Marenostrium 4. The system is composed of two main elements, the model compiler libraries and the simulator. The XML file is read with the *pugixml* c code library; from this XML, an intermediate structure is generated with the *sema.c* class, which will be the basis for the *codegen.c* class to generate the executable C code in the Marenostrium 4 owned by the BSC-CNS.

### 3.3. Model-Compiler Libraries

These libraries are responsible for translating the XML files representing an SDL model into a C++ source file that can be integrated into the simulator. The execution of the libraries consists of three phases. The first phase reads all the information from the root file of the project and gathers all the information needed by the next phase. This phase performs necessary checks, e.g., ensuring that all required files exist.

The second phase performs additional checks to ensure the model is semantically correct and generates additional information about the model needed by the final phase. For example, this phase determines which PROCESSES can communicate among themselves, what SIGNALS will be sent and received by each process, what STATES do not receive any SIGNAL defined on the model (by an OUTPUT), and which PROCESS may potentially create other PROCESSES during the execution. This information will be used by the simulator to guarantee correct executions and to optimize the parallelization of the code. This phase also validates the consistency and completeness of the model and reports any errors or warnings to the user.

The third phase is responsible for transforming all the information about the model gathered through the previous phases into a C++ source file. This phase generates the code that implements the structure and behavior of the model, using classes, methods, variables, and statements. The code also includes the necessary libraries and directives for parallelization, such as OpenMP [3,4,47]. Figure 6 shows a simplified version of the code generated for *PManager*, which is a process that manages the creation and deletion of other processes in the model.

```

1 struct Bprime_PManager : public Process {
2     int N = 100;
3     int P = 3;
4     ...
5     enum State {
6         END,
7         PREPARING,
8     };
9     State state;
10    Process_Instance num_instance;
11    // Process initialization function
12    void init();
13    // Process event handler function
14    void specific_step(shared_ptr<Event>& e);
15    static int total_instances;
16 };
17 int Bprime_PManager::total_instances = 0;
18
19 void Bprime_PManager::init() {
20     sqrtN=sqrt(N);
21     S = (N-(sqrtN+1)) / P;
22     R = (N-(sqrtN+1)) % P;
23     L = sqrtN + i*S + 1;
24     H = L+S-1 + (i < R);
25
26     shared_ptr<NEW_SEGMENT_event>
27     send_event(new NEW_SEGMENT_event);
28     send_event->tsrc = time; send_event->delay = 1;
29     // Send NEW_SEGMENT to SELF
30     sim_send(this, send_event);
31
32     state = PREPARING;
33 }

```

```

34 void Bprime_PManager::specific_step(shared_ptr<Event>& e) {
35     switch (state) {
36     case END:
37         switch (e->type) {
38         case Event_type::PARTIAL_RESULTS:
39             shared_ptr<PARTIAL_RESULTS_event> e1 = e;
40             list2 = e1->list2; H = e1->H; L = e1->L;
41
42             ReportList2(list2, c, H, L);
43
44             state = END;
45             break;
46         }
47     case PREPARING:
48         switch (e->type) {
49         case Event_type::NEW_SEGMENT:
50             shared_ptr<NEW_SEGMENT_event> e1 = e;
51             if (i<P) {
52                 {
53                     shared_ptr<SEGMENT_event> send_event(new SEGMENT_event);
54                     // Timestamp = tsrc + delay
55                     send_event->tsrc = time; send_event->delay = 1;
56                     send_event->type = Event_type::SEGMENT;
57                     // Setup event data to send
58                     send_event->i = i; send_event->L = L;
59                     send_event->H = H; send_event->P = P;
60                     send_event->R = R;
61                     // Send to PSegment instance '\\i\\'
62                     auto p = proc_instances[PKind::Bprime_PSegment][i];
63                     sim_send(p, send_event);
64                 }
65                 ...
66                 // Send SKIP and NEW_SEGMENT events
67                 ...
68                 i++;
69                 L = H+1;
70                 H = L+S-1 + (i < R);
71                 state = PREPARING;
72             } else {
73                 state = END;
74             }
75             break;
76         }
77     }
78     break;
79 }
80 }
81 }

```

Figure 6. PManager C++ output code (simplified).

### 3.4. Simulator

This tool implements two compatible simulation environments able to execute the model in the C++ source file: one sequential and one parallel. The sequential simulation runs the simulation without any use of parallelism. On the other hand, the parallel implementation can run parts of the simulation concurrently by using OpenMP [47] if several conditions are met.

Each process  $P$  has one external event queue  $Q_R$  for each process  $R$  connected to  $P$ .  $P$  also has an additional event queue  $Q_{\text{self}}$  for events it sends to itself. A process  $P$  can send events to other processes, but these events will have a timestamp greater or equal to the timestamp of the last event that was received by  $P$ ; we call this timestamp,  $L_P$ . Moreover, the sequence of events sent from  $P$  to  $R$  must have nondecreasing timestamps. For each external event queue  $Q_K$  in  $P$ , there is one or more events with a timestamp  $M_K$  that is larger or equal to the other events. We define the critical time  $T_{\text{crit}}$  of a process  $P$  as the minimum of all the  $M_K$  timestamps. A process  $P$  can be simulated concurrently with other processes if it has events in its queues whose timestamp is lower than  $T_{\text{crit}}$ . Figure 7 shows an example of this process.

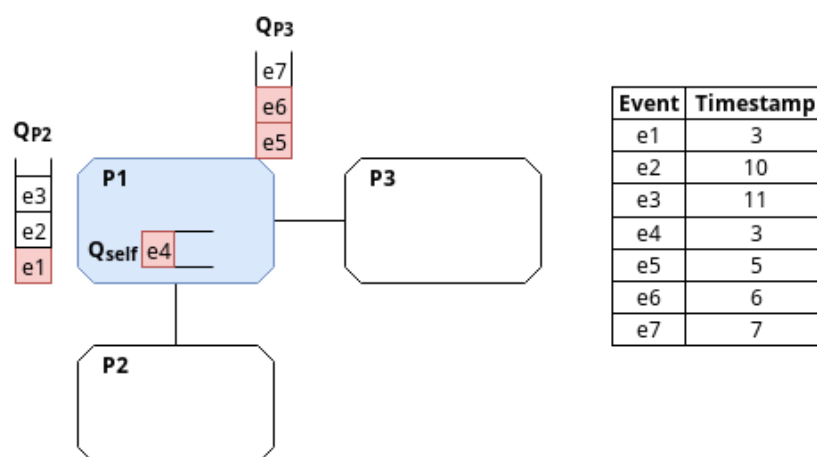


Figure 7. Example of parallel process.

Process P1 is connected to P2 and P3 and has events to be consumed. From P2 it has events e1, e2, and e3 with timestamps 3, 10, and 11, respectively, whereas from P3 it has e5, e6, and e7 with timestamps 5, 6, and 7. In addition, P1 has an event, e4, sent to itself with timestamp 3.

The  $T_{crit}$  of P1 is  $\min(e3, e7) = 7$ . In this example, P1 can execute, in parallel, the events with time e1, e4, e5, e6 marked as red. That is because P2 and P3 only can send events with timestamps greater equal 11 and 7, respectively.

To be able to detect this situation, the parallel simulator has an inspection and execution phase. The former goes through the current list of processes in the system, gathering the ones able to run in parallel while searching the process owning the minimum timestamp event. The latter runs in parallel with the processes gathered, in this case, more than one has been found. If not, the process with the minimum timestamp event is executed.

#### 4. Case Studies: Numerical and Agent-Based Models

For verifications of the HPC library developed to directly execute SDL models in HPC environments, we replicated two standard simulation models: a numerical simulation and an agent-based model. The objective of the former, the *Sieve of Eratosthenes model*, is to verify the implementation of the library and to test the robustness of the process. Second, the agent-based model is a replication of a seminal in social sciences *Artificial Anasazi model*. Here, it is used to test the potential of the proposed workflow to facilitate communication in a multidisciplinary team. The hope is that the SDL language can become a bridge between scientists and engineers working on complex simulation models.

The Artificial Anasazi model [18,48] is a well-known model within the agent-based modeling community. It describes the population dynamics of the Ancestral Pueblo People, who lived in the Long House Valley, Arizona between AD 800 and AD 1350. The model simulates population fluctuations and spatial distribution of human settlements on the landscape as a result of climatic shifts and individual decision-making processes.

The aim of using this highly complex agent-based model was to test the viability of using the SDL language as the ‘communication platform’ with a highly challenging context likely to mirror real-world circumstances. The use of a graphical language like SDL, as a communication platform or mechanism, simplifies the interaction between the different specialists who are involved in the validation and verification of the models that will eventually be executed on the platform to support decisions. Throughout the modeling process, a social simulation expert collaborated closely with computer science engineers to translate the model’s ontology, develop the model’s implementation, and solve the barriers preventing the efficient parallelization of the code and other issues that needed to be overcome for the model to run on a HPC environment. The discussion between the context expert (the social scientist) and the HPC experts became very productive and straightforward since they all shared the same basis: the SDL model. The SDL model also helped to ensure the



consistency and correctness of the model, and to avoid the loss of information or meaning during the translation from the conceptual to the computational level.

In the next section, we will focus on the SDL Sieve of Eratosthenes model to showcase the functionality of the presented framework.

#### 4.1. Sieve of Eratosthenes SDL Model

The prime number model is based on a parallel version of the Sieve of Eratosthenes method to calculate the prime numbers on a segment. To do so, an important element is to detail the structure of the model, since this structure will facilitate the parallelization of the algorithm in the HPC environment. Figure 8 shows the structure of the model, comprising the SYSTEM and one BLOCK “Bprime”. The Bprime BLOCK (Figure 9) contains the three main PROCESSES that will be used to calculate the prime numbers, “PCreator” (Figure 10), “PManager” (Figure 11), and “PSegment” (Figure 12).

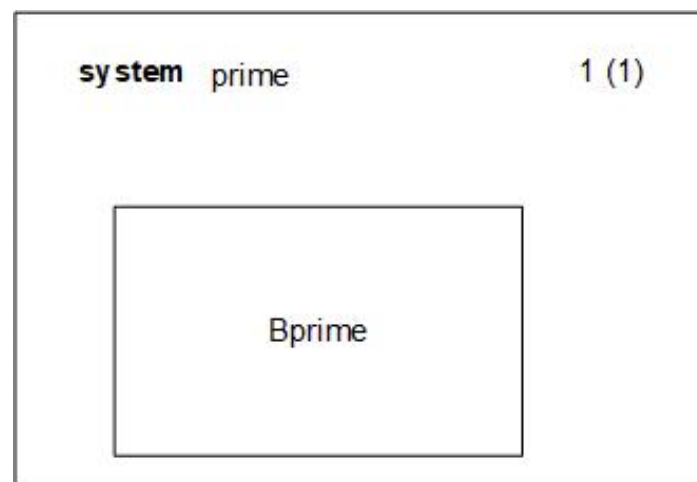


Figure 8. Definition of the structure of the prime model, SYSTEM diagram.

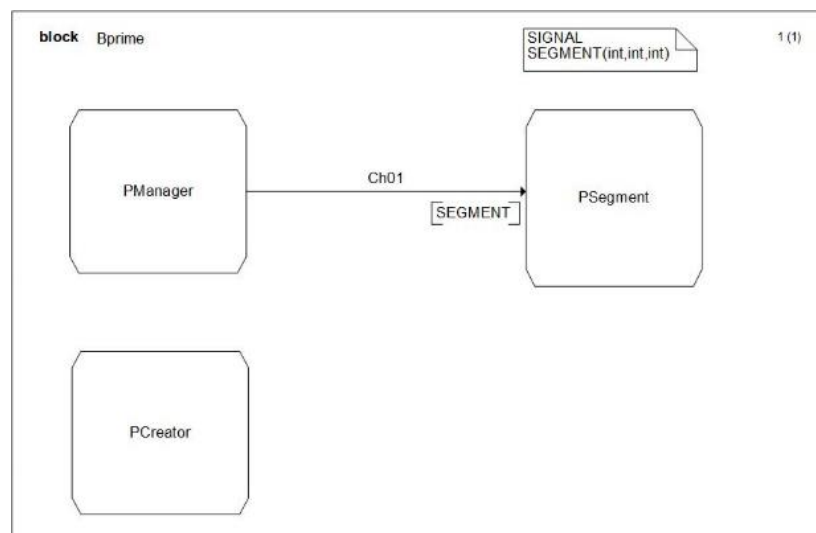
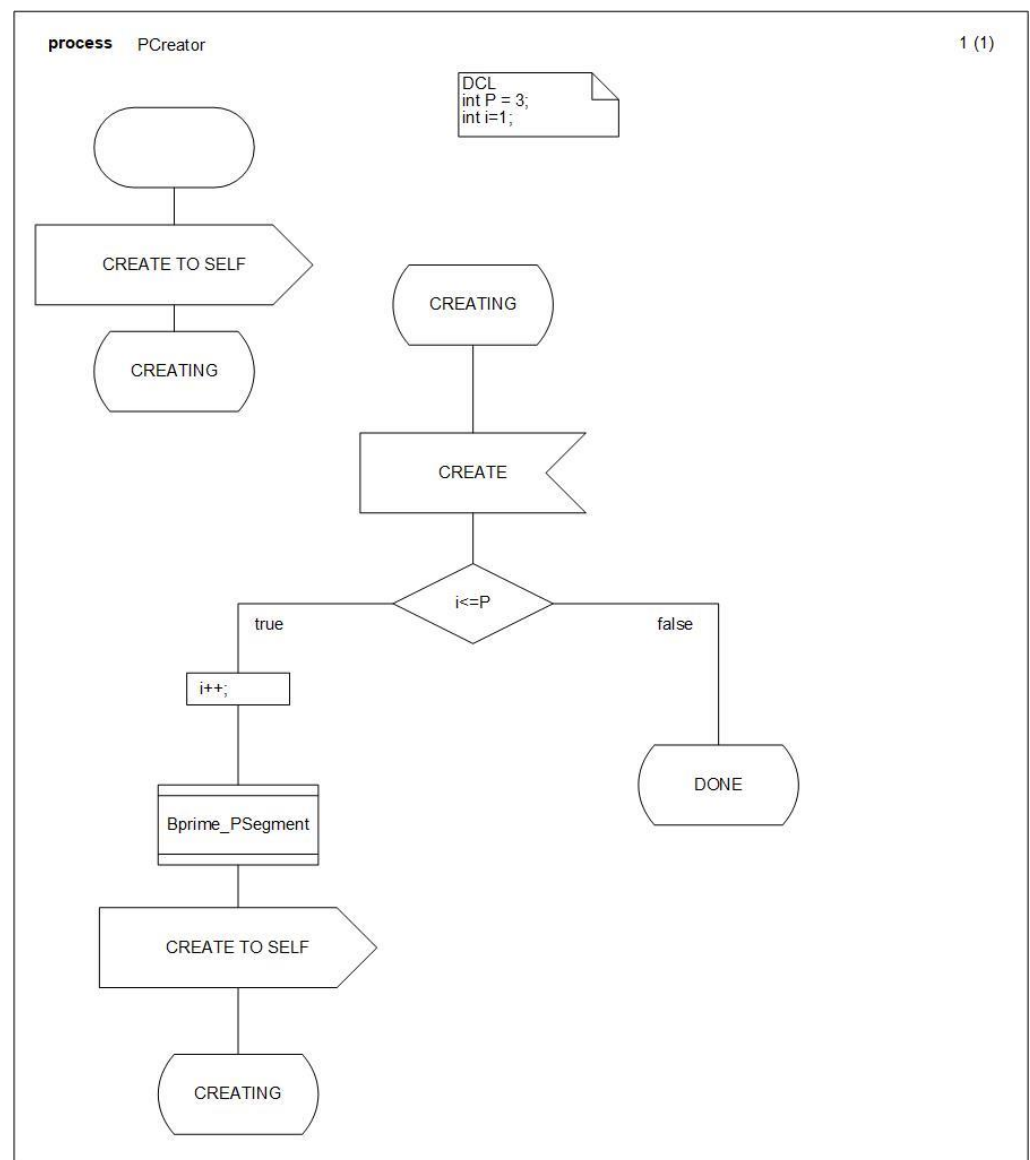


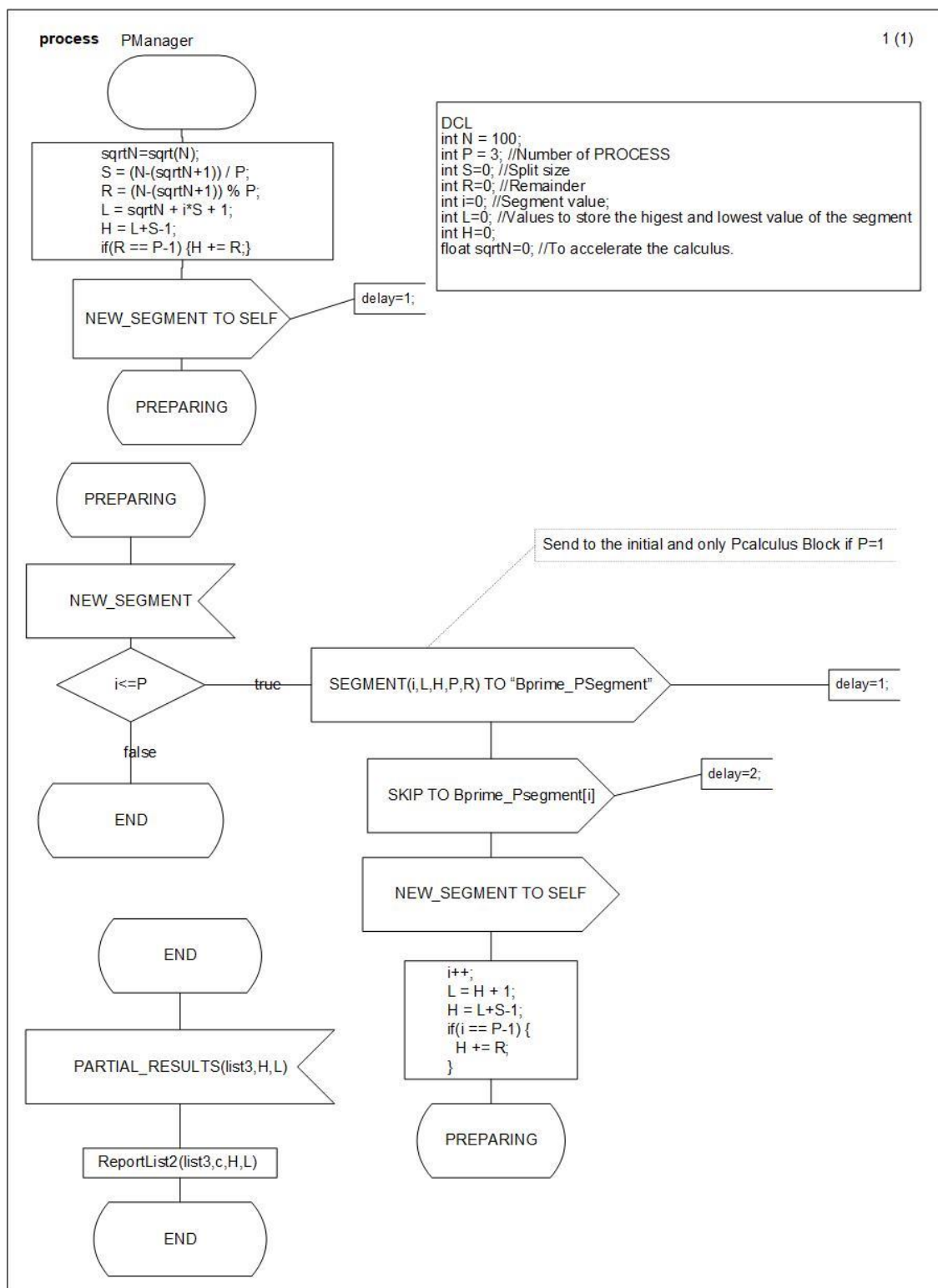
Figure 9. Definition of the structure of the prime model, BLOCK Bprime.



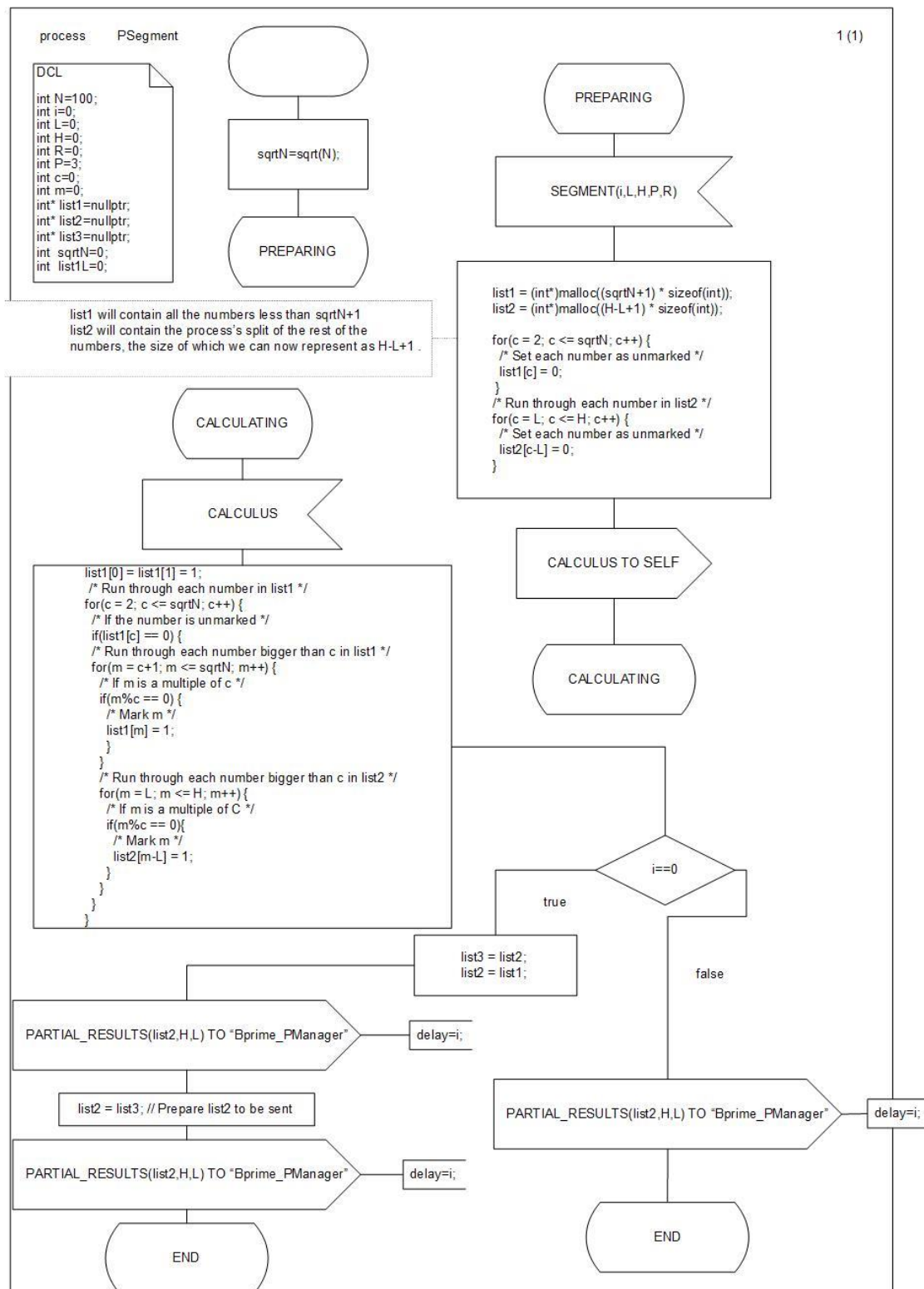
**Figure 10.** *PCreator* PROCESS. This PROCESS defines the parallelization level of the Sieve of Eratosthenes approach. Parameter P represents the number of parallel instances that will be executed in parallel; in this example, there are 3.

From this SDL representation of our approach to the parallel Sieve of Eratosthenes problem, SDLPS generates an XML representation of the model that will be used as an input for the libraries, see Figure 13.

This XML representation of the SDL model allows us to simplify the interaction between the different tools that must take care of the model. We prefer to use this over SDL/PR because this XML representation simplifies working with it in our computational frameworks.



**Figure 11.** *PManager* PROCESS. On this PROCESS are prepared all the segments *Psegment* that will be used for the distributed calculus and generate the results from all the different *Psegment* that are involved in the parallel calculus.



**Figure 12.** PSegment PROCESS. This PROCESS is defined as the main calculus (on the first task), and is sent the partial result, calculated on this segment to the PManager that will store all the results calculated.

```

1 <?xml version="1.0" encoding="UTF-16"?>
2 <sdprocess>
3   <process id="1" name="PManager" IP="" portRead="" implementation="" inherits="">
4     <DCLS>...</DCLS>
5     <procedures>
26       <start>
27         <task id="1" name="">sqrtN=sqrt(N); S = (N-(sqrtN+1)) / P; R = (N-(sqrtN+1)) % P; L = sqrtN + i*S + 1; H = L+5-1; if(R == P-1) {H += R;}</task>
28         <output id="2" name="SEGMENT" self="" to="PSegment" via="">
29           <param name="" value="">
30             <userparam name="i" value="i"/>
31             <userparam name="L" value="L"/>
32             <userparam name="H" value="H"/>
33             <userparam name="P" value="P"/>
34             <userparam name="R" value="R"/>
35           </output>
36         <decision id="3" name="" iftrue="6" iffalse="4">i<P<decision>
37         <output id="4" name="START_CALCULUS" self="" to="" via="">
38         <setstate id="5" name="READY"/>
39         <create id="6" name="PSegment"/>
40         <task id="7" name="">i++; L = sqrtN + i*S + 1; H = L+5-1;</task>
41         <output id="8" name="SEGMENT" self="" to="Pprime_PSegment" via="">
42           <userparam name="i" value="i"/>
43           <userparam name="L" value="L"/>
44           <userparam name="H" value="H"/>
45           <userparam name="P" value="P"/>
46           <userparam name="R" value="R"/>
47         </output>
48         <output id="9" name="NEW_SEGMENT" self="" to="" via="">
49         <setstate id="10" name="PREPARING"/>
50       </start>
51       <state name="PREPARING">...</state>
52     </procedures>
53   </process>
54 </sdprocess>
55

```

**Figure 13.** XML representation obtained through SDLPS to represent the *PManager* PROCESS of the prime model.

#### 4.2. Sieve of Eratosthenes Parallel Execution

To take advantage of the parallelism condition of the simulator, the model follows four well-defined phases.

First, the *PCreator* creates PROCESS *PSegment* and gets it ready to work. Second, the *PManager* distributes the chunks of the sieve to all *PSegments* sending the boundaries for each one. To satisfy the parallelism, condition a SKIP message is also sent with a higher timestamp. This will make the simulator aware that all *PSegments* can run in parallel and execute them. Third, *PSegments* compute the partial sieve and send the results to *PManager*. To ensure the results are printed in order, each *PSegment* sends the data with a timestamp plus a delay equal to its process instance. Finally, the *PManager* processes each message sequentially, printing the results.

An MN4 job script is a text file that contains the instructions and parameters for running a job on the Marenosturm 4 supercomputer [49]. A job is the execution unit for Slurm, which is the utility used for batch processing support on Marenosturm 4. An MN4 job script typically includes the following elements: (i) the shebang line (`#!/usr/bin/env bash`), which indicates the shell to use for the script; (ii) the SBATCH directives (`#SBATCH`), which specify the job name, output and error files, number of tasks and CPUs per task, time limit, and reservation name for the job, and these directives are preceded by a hash sign (`#`) and are read by Slurm before executing the script; (iii) the module load command (`module load`), which loads the required software modules for the job, such as compilers, libraries, or tools; (iv) the environment variables (`export`), which set the values of certain variables for the job, such as the number of threads or the path to Extrae; and (v) the commands (`srun`, `mpirun`, etc.), which launch the executable or application for the job, with optional arguments or options. Figure 14 shows an example of the MN4 job script used to execute the model.

```

1 #!/bin/bash
2 #SBATCH --job-name=primers
3 #SBATCH --workdir=.
4 #SBATCH --output=primers_%j.out
5 #SBATCH --error=primers_%j.err
6 #SBATCH --cpus-per-task=48
7 #SBATCH --ntasks=1
8 ./primers

```

**Figure 14.** Example of MN4 job script.



Once our job is executed, we can obtain the results in the output files. This is the result obtained from computing the sieve between 2 and 1000 using 5 PSegments, see Figure 15.

```
$ cat primers_6952114.out
NOTE: 5 can run in parallel
NOTE: 5 can run in parallel
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163
167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347
349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541
547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739
743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953
967 971 977 983 991 997
```

Figure 15. Example of MN4 sieve results.

The SDLPS can be downloaded at <https://sdlps.com>, accessed on 15 November of 2023.

## 5. Discussion

The examples we have shown here are simple enough in order to understand the details of the code generation mechanism we use for the Marenostrom. However, this approach makes sense when we develop projects where personnel with different backgrounds must collaborate to define a complex model. An example of this is the simulation of the current pandemic situation in Catalonia, see [50], where the modeling using SDL largely simplifies the interaction between the different actors (specialists) that are involved in the project. Other examples where this approach has been used are in the construction sector [28,44], where the model connects with the energyPlus [51] engine to optimize building sustainability for the simulation of slap avalanches [43], software [52–54], or the simulation of industrial systems [38]. The capability to generate the code that can later be executed on Marenostrom allows us to avoid the constraint related to the computational capabilities of the infrastructure we will use. Moreover, this approach enables us to explore different scenarios and parameters for the simulation, such as the impact of vaccination, lockdowns, social distancing, and testing strategies. By using SDL, we can easily modify and validate the model according to the latest data and evidence. This way, we can provide useful insights and recommendations for policymakers and health authorities to manage the pandemic situation in Catalonia.

Some existing proposals allow the definition of simulation models to be executed in a HPC environment. For example, AnyLogic is a tool that supports different modeling paradigms, such as agent-based, discrete event, and system dynamics. It allows the user to create graphical models using drag-and-drop elements, and to define the logic and behavior of the model using Java code. AnyLogic can also export the models to standalone applications or cloud platforms, and run them on a HPC environment, using MPI (Message Passing Interface) or Spark libraries [55]. However, this proposal relies on a proprietary platform and not on a standard formal language like SDL. SDL allows an agnostic representation of the simulation model that seeks to be independent of the final tool or platform that will be used to perform the implementation. This approach also enables the user to validate and verify the model before translating it into executable code that can run on different environments.

Some limitations of our approach are (i) we generated the code using OpenMP libraries that are focused on Marenostrom HPC architecture. Some adaptation to the C libraries that generate this code must be performed to ensure that it works on other HPC platforms. (ii) Not all the features of SDL language are currently supported. Although SDLPS supports the use of Cellular Automaton structures, SDLPS for HPC does not allow yet the use of the dynamic creation of new agents using the CREATE SDL BLOCK. This limitation implies the use of data matrices or Cellular Automaton to generate MAS (Multi-Agent Systems) in case they are needed. We are developing the necessary structures to enable the use of this BLOCK shortly. However, we want to remark that the models do not need to be changed since, being represented by SDL, and being SDL a standard language, it assures compatibility between other HPC platforms that also understand SDL language or local

computers that can execute SDLPS in common PC, or other programs like PragmaDEV [33] that also understands SDL.

When the power of a supercomputer is needed, the code generation mechanism presented in this paper ensures that the model is translated into efficient and scalable code that can run on Marenostrium and produce reliable and timely results. This mechanism is based on the standard language SDL, which is a formal, graphical, unambiguous, and complete modeling language for complex systems. By using SDL, the modelers can focus on the logic and behavior of the system, without worrying about the low-level details of the code. The code generation mechanism automatically translates the SDL model into C++ code that can be compiled and executed on Marenostrium or other HPC platforms. The generated code is optimized for parallelism and performance, and can take advantage of the advanced features of Marenostrium, such as its high-speed network, large memory capacity, and heterogeneous architecture. The code generation mechanism also supports debugging and verification of the SDL model, as well as integration with other tools and libraries. Thus, the code generation mechanism provides a convenient and effective way to leverage the power of Marenostrium for complex system simulation using SDL.

As a future work, we are planning to develop the CREATE SDL BLOCK to allow the dynamic creation of SDL AGENTS, like PROCESS or BLOCKS, simplifying the implementation and definition of MAS models. Also, we plan to develop a web interface, on <https://sdlps.com> (accessed on 15 November of 2023) that allows the parametrization of the SDLPS models and also the generation of the code for execution in different platforms.

## 6. Concluding Remarks

In this paper, we presented a set of tools focused on the optimization of runtime and the compatibility of code libraries with the standard language SDL, which aims to increase the parallelization capacity of simulation models defined in SDL. This framework has the potential to significantly increase the attractiveness and usability of HPC for end users, especially those with limited HPC capacities and low-to-moderate levels of HPC expertise. The integration of social science models through formal languages may extend the target audience of the HPC infrastructures to an important group of end-users who are currently not taking full advantage of its capacities due to the lack of necessary know-how or limited access to experts and resources. A more comprehensive application of SDL (a formal, graphical, unambiguous, and complete modeling language) has the potential for streamlining, automating, and documenting the process of simulation design and development among non-academics and within academic fields where the level of computational expertise is low (e.g., social sciences, humanities). We focused on two case studies performed in the framework of the “HPC optimization of SDLPS distributed simulator” PRACE project. However, the methodology applied here can be used for any project where non-technical stakeholders need to be involved in the definition of the model.

The benefits of the proposed pipeline are twofold. On the one hand, SDL allows for the definition of complex simulation models and shows a high potential for seamless integration into IoT ecosystems, generating AIoT (Artificial Intelligence of Things) environments. On the other hand, it automatically translates conceptual models into code that can be executed in a HPC infrastructure. Thus, it gives interested stakeholders the ability to solve real-world problems by quickly designing and implementing complex models and running them using HPC tools, without the need for an extensive team of computer science experts. The hope is that the SDL-based pipeline combined with the presented simulation will open a new avenue to the HPC world for a currently underrepresented set of users, providing them with a natural access point to this state-of-the-art technology.

**Author Contributions:** Conceptualization, P.F.i.C. and I.R.; methodology, P.F.i.C.; software, P.F.i.C. and J.G.i.S.; validation, P.F.i.C., I.R. and J.G.i.S.; formal analysis, P.F.i.C. and J.G.i.S.; investigation, P.F.i.C. and I.R.; resources, P.F.i.C. and I.R.; data curation, P.F.i.C. and J.G.i.S.; writing—original draft preparation, P.F.i.C.; writing—review and editing, P.F.i.C. and I.R.; visualization, P.F.i.C.; supervision,

P.F.i.C. and I.R.; project administration, P.F.i.C. and I.R.; funding acquisition, P.F.i.C. and I.R. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research has been founded by the Project PRACE (Partnership for Advanced Computing in Europe) titled “HPC optimization of SDLPS distributed simulator”.

**Data Availability Statement:** Data are contained within the article.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. IBM. What Is HPC? *Introduction to High-Performance Computing*. Available online: <https://www.ibm.com/topics/hpc> (accessed on 5 November 2023).
2. Qian, D. High performance computing: A brief review and prospects. *Natl. Sci. Rev.* **2016**, *3*, 16. [CrossRef]
3. Bak, S.; Bertoni, C.; Boehm, S.; Budiardja, R.; Chapman, B.M.; Doerfert, J.; Eisenbach, M.; Finkel, H.; Hernandez, O.; Huber, J.; et al. OpenMP application experiences: Porting to accelerated nodes. *Parallel Comput.* **2022**, *109*, 102856. [CrossRef]
4. Hoffmann, R.B.; Löff, J.; Griebler, D.; Fernandes, L.G. OpenMP as runtime for providing high-level stream parallelism on multi-cores. *J. Supercomput.* **2022**, *78*, 7655–7676. [CrossRef]
5. Salloum, S.; Dautov, R.; Chen, X.; Peng, P.X.; Huang, J.Z. Big data analytics on Apache Spark. *Int. J. Data Sci. Anal.* **2016**, *1*, 145–164. [CrossRef]
6. Jiang, M.; Gallagher, B.; Chu, A.; Abdulla, G.; Bender, T. Exploiting Spark for HPC Simulation Data: Taming the Ephemeral Data Explosion. In Proceedings of the HPCAsia2020: International Conference on High Performance Computing in Asia-Pacific Region, Fukuoka, Japan, 15–17 January 2020; ACM International Conference Proceeding Series. pp. 150–160. [CrossRef]
7. Castañé, G.G.; Xiong, H.; Dong, D.; Morrison, J.P. An ontology for heterogeneous resources management interoperability and HPC in the cloud. *Future Gener. Comput. Syst.* **2018**, *88*, 373–384. [CrossRef]
8. Faheem, H.M.; König-Ries, B.; Aslam, M.A.; Aljohani, N.R.; Katib, I. Ontology design for solving computationally-intensive problems on heterogeneous architectures. *Sustainability* **2018**, *10*, 441. [CrossRef]
9. Böhm, S.; Běhálík, M. Usage of Petri Nets for high performance computing. In Proceedings of the FHPC’12—2012 ACM SIGPLAN Functional High Performance Computing, Copenhagen, Denmark, 15 September 2012; pp. 37–47. [CrossRef]
10. Jensen, K. *Coloured Petri Nets*; Monographs in Theoretical Computer Science An EATCS Series; Springer: Berlin/Heidelberg, Germany, 1997. [CrossRef]
11. Liao, C.; Lin, P.-H.; Verma, G.; Vanderbruggen, T.; Emani, M.; Nan, Z.; Shen, X. HPC Ontology: Towards a Unified Ontology for Managing Training Datasets and AI Models for High-Performance Computing. In Proceedings of the 2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC), St. Louis, MO, USA, 15 November 2021; pp. 69–80. [CrossRef]
12. Sherratt, E. SDL: Meeting the IoT challenge. In *System Analysis and Modeling*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2016; Volume 9959, pp. 36–50. [CrossRef]
13. Sherratt, E. SDL in a changing world. In *System Analysis and Modeling*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2005; Volume 3319, pp. 96–105. [CrossRef]
14. Sherratt, E.; Loftus, C. Designing distributed services with SDL. *IEEE Concurr.* **2000**, *8*, 59–66. [CrossRef]
15. Sherratt, E.; Ober, I.; Gaudin, E.; Casas, P.F.I.; Kristoffersen, F. SDL—The IoT Language. In *SDL 2015: Model-Driven Engineering for Smart Cities*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9369, pp. 27–41. [CrossRef]
16. Ellsberger, J.; Hogrefe, D.; Sarma, A. *SDL: Formal Object-Oriented Language for Communicating Systems*, 2nd ed.; Prentice Hall: Upper Saddle River, NJ, USA, 1997.
17. Meertens, L. Functional Pearl Calculating the Sieve of Eratosthenes. *J. Funct. Program.* **2004**, *14*, 759–763. [CrossRef]
18. Janssen, M.A. Understanding Artificial Anasazi. *JASSS* **2009**, *12*, 13.
19. Trabes, G.G.; Wainer, G.A.; Gil-Costa, V. A Parallel Algorithm to Accelerate DEVS Simulations in Shared Memory Architectures. *IEEE Trans. Parallel Distrib. Syst.* **2023**, *34*, 1609–1620. [CrossRef]
20. Concepcion, A.I.; Zeigler, B.P. DEVS formalism: A framework for hierarchical model development. *IEEE Trans. Softw. Eng.* **1988**, *14*, 228–241. [CrossRef]
21. Zeigler, B.P.; Song, H.S.; Kim, T.G.; Praehofer, H. DEVS framework for modelling, simulation, analysis, and design of hybrid systems. In *Hybrid Systems II*; Springer: Berlin/Heidelberg, Germany, 1995; pp. 529–551. [CrossRef]
22. Werner, T.; Päßler, C.; Richter, M.; Kabadshow, I.; Werner, M. A Petri-Net-Based Approach to Modeling Communication Algorithms for HPC Molecular Dynamics Simulations. In Proceedings of the PNSE’23: International Workshop on Petri Nets and Software Engineering, Lisbon, Portugal, 26–27 June 2023; Available online: <http://ceur-ws.org> (accessed on 31 October 2023).
23. Li, Z.; Jiao, L.; Hu, X. Performance analysis for job scheduling in hierarchical HPC systems: A coloured petri nets method. In *Algorithms and Architectures for Parallel Processing*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2015; Volume 9531, pp. 259–280. [CrossRef]

24. Wang, J. Petri Nets for Dynamic Event-Driven System Modeling. In *Handbook of Dynamic System Modeling*; Fishwick, P.A., Ed.; Chapman & Hall: Gainesville, FL, USA, 2007; pp. 24-1–24-17. [\[CrossRef\]](#)
25. Cabasino, M.P.; Giua, A.; Seatzu, C. Introduction to petri nets. In *Control of Discrete-Event Systems*; Lecture Notes in Control and Information Sciences; Springer: London, UK, 2013; Volume 433, pp. 191–211. [\[CrossRef\]](#)
26. Doldi, L. *SDL Illustrated—Visually Design Executable Models*, 1st ed; TMSO Systems: University Park, PA, USA, 2001.
27. ITU-T. ITU-T-2019—Specification and Description Language—Overview of SDL-2010, ITU-T Recommendation Z.100. 2019. Available online: <http://handle.itu.int/11.1002/1000/14048> (accessed on 31 October 2023).
28. Casas, P.F.I.; Casas, A.F.I.; Garrido-Soriano, N.; Ortiz, J.; Casanovas, J.; Salom, J. Optimal Buildings' Energy Consumption Calculus through a Distributed Experiment Execution. *Math. Probl. Eng.* **2015**, *2015*, 267974. [\[CrossRef\]](#)
29. Fonseca Casas, P. Transforming classic Discrete Event System Specification models to Specification and Description Language. *Simulation* **2015**, *91*, 249–264. [\[CrossRef\]](#)
30. Vangheluwe, H.L.M. DEVS as a common denominator for multi-formalism hybrid systems modelling. In Proceedings of the CACSD—IEEE International Symposium on Computer-Aided Control System Design (Cat. No. 00TH8537), Anchorage, AK, USA, 25–27 September 2000.
31. Zeigler, B.P.; Praehofer, H.; Kim, T.G. *Theory of Modeling and Simulation Handbook of Simulator-Based Training Creating Computer Simulation Systems: An Introduction to the High Level Architecture*; Prentice Hall: Upper Saddle River, NJ, USA, 2000; Volume 100. [\[CrossRef\]](#)
32. Doldi, L. *Validation of Communications Systems with SDL: The Art of SDL Simulation and Reachability Analysis*; Wiley & Sons: Hoboken, NJ, USA, 2003.
33. PragmaDev SARL. PragmaDev Studio. Available online: <http://www.pragmadev.com/product/index.html> (accessed on 9 January 2016).
34. PragmaDev. *Graphical Language to Specify and Design Real Time and Embedded Software September*; PragmaDev: Paris, France, 2013.
35. Cinderella ApS. Cinderella. 2011. Available online: <http://www.cinderella.dk/> (accessed on 9 January 2016).
36. Rauchwerger, Y.; Kristoffersen, F.; Lahav, Y. Cinderella SLIPPER: An SDL to C-code generator. In *SDL 2005: Model Driven*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3530, pp. 210–223. [\[CrossRef\]](#)
37. IBM Co. Rational SDL Suite. Available online: <http://www-03.ibm.com/software/products/en/ratisdlsuit> (accessed on 9 January 2016).
38. Casas, P.F.I.; Palomés, X.P.; Garcia, J.C.; Jové, J. Definition of virtual reality simulation models using specification and description language diagrams. In *SDL 2013: Model Driven Dependability Engineering*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 258–274. [\[CrossRef\]](#)
39. Sargent, R.G. Verification validation and accreditation of simulation models. In Proceedings of the 2000 Winter Simulation Conference Proceedings (Cat. No. 00CH37165), Orlando, FL, USA, 10–13 December 2000; pp. 50–59.
40. Balci, O. Verification, Validation, and Certification of Modeling and Simulation Applications. In Proceedings of the 2003 Winter Simulation Conference, New Orleans, LA, USA, 7–10 December 2003; pp. 150–158.
41. Casas, P.F.I. A Continuous Process for Validation, Verification, and Accreditation of Simulation Models. *Mathematics* **2023**, *11*, 845. [\[CrossRef\]](#)
42. Sargent, R.G. Verification and validation of simulation models. *J. Simul.* **2013**, *7*, 12–24. [\[CrossRef\]](#)
43. Fonseca, P.; Colls, M.; Casanovas, J. A novel model to predict a slab avalanche configuration using m:n-CAk cellular automata. *Comput. Environ. Urban Syst.* **2011**, *35*, 12–24. [\[CrossRef\]](#)
44. Casas, P.F.I.; Casas, A.F.I. NECADA. Optimization software for sustainable architecture. In *Building Simulation Conference*; IBPSA: Hyderabad, India, 2015.
45. IEEE. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)—Framework and Rules. Available online: <https://standards.ieee.org/ieee/1516/3744/> (accessed on 25 September 2023).
46. BSC. Marenostrom. Available online: <https://www.bsc.es/es/marenostrom/marenostrom> (accessed on 6 June 2019).
47. OpenMP. OpenMP. Available online: <https://www.openmp.org/> (accessed on 6 June 2019).
48. Diamond, J.M. Life with the artificial Anasazi. *Nature* **2002**, *419*, 567–568. [\[CrossRef\]](#)
49. BSC. Running Jobs | BSC Support Knowledge Center. Available online: <https://www.bsc.es/supportkc/docs/MareNostrum4/slurm/> (accessed on 25 September 2023).
50. Casas, P.F.I.; Subirana, J.G.I.; Carrasco, V.G.I.; Palomes, X.P.I.; Wainer, G. Formal Modeling and Simulation for SARS-CoV-2 Containment Scenarios in Catalonia. *Comput. Sci. Eng.* **2022**, *24*, 86–90. [\[CrossRef\]](#)
51. Reference, T.E.; Input, E. *Input Output Reference: The Encyclopedic Reference to EnergyPlus Input and Output, version 8.0*; Big Ladder Software: Denver, CO, USA, 2014.
52. Podnar, I.; Mikac, B.; Caric, A. SDL based approach to software process modeling. In *Software Process Technology*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2000; Volume 1780, pp. 190–202. [\[CrossRef\]](#)
53. Díaz, M.; Garrido, D.; Troya, J.M. Development of distributed real-time simulators based on CORBA. *Simul. Model. Pract. Theory* **2007**, *15*, 716–733. [\[CrossRef\]](#)

54. Dragomir, I.; Redondo, C.; Jorge, T.; Gouveia, L.; Ober, I.; Kolesnikov, I.; Bozga, M.; Perrotin, M. Model-checking of space systems designed with TASTE/SDL. In Proceedings of the ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022: Companion Proceedings, Montreal, QC, Canada, 23–28 October 2022; pp. 237–246. [\[CrossRef\]](#)
55. Borshchev, A.; Karpov, Y.; Kharitonov, V. Distributed simulation of hybrid systems with AnyLogic and HLA. *Future Gener. Comput. Syst.* **2002**, *18*, 829–839. [\[CrossRef\]](#)

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.