An Overview of the Quartz Modelling and Simulation Framework

Romain Franceschini[1], Paul-Antoine Bisgambiglia[1], Paul Bisgambiglia[1] and David R. C. Hill[2]
[1]*University of Corsica, UMR 6134, Campus Grimaldi, Corte, France*
[2]*Clermont Auvergne University, UMR 6158, ISIMA LIMOS, F-63173 Aubiere, France*

Keywords: Simulation Environment, Discrete-event Simulation, PDEVS, DPDEMAS, multiPDEVS.

Abstract: Quartz is a modelling and simulation framework enabling the development and the execution of models based on the Parallel Discrete Event System Specification (PDEVS) formalism. In this paper we give an overview of the tool by giving details on its design and features. An experimental comparison shows that Quartz yields performances comparable to what is observed with aDEVS, one of the most efficient tool implemented with a compiled language. We present our efforts to provide dedicated expressions and verification facilities to the modeler and give examples of application domains that motivated this software. Namely, agent-based modeling through the Dynamic Parallel Discrete Event Multi-Agent (DPDEMAS) specification and multicomponent modeling with the multiPDEVS formalism.

1 INTRODUCTION

One among the many goals of the modelling and simulation field is to provide tools to domain experts so they may design and execute their models in a convenient way. In this context, an important number of simulation framework based on the Discrete Event System Specification (DEVS) formalism (Zeigler et al., 2000) or its parallel variant (PDEVS) have been proposed, *e.g.* VLE (Quesnel et al., 2009), PyPDEVS (Van Tendeloo and Vangheluwe, 2014). The DEVS formalism is a specification language which offers a precise description of the various entities involved in the modeling and simulation process.

In this paper we give an overview of Quartz, a modelling and simulation framework based on PDEVS (Chow, 1996). Its development is driven by the following concerns: (1) allow the inclusion of the framework in a larger simulation environment, (2) offer acceptable performances in comparison with other PDEVS-based simulators and (3), adopt an architecture which facilitates the addition of new PDEVS variant formalisms.

Among available simulation tools, three groups can be established according to their strategy. The first allows full model compilation with its associated simulators all into an optimized executable. Consequently, models must be programmed in a technical languages such as C++, as with the Virtual Laboratory Environment (Quesnel et al., 2009), *a.k.a* VLE, or with A Discrete EVent System simulator (Nutaro, 1999), *a.k.a* aDEVS. Conversely, a second group of simulators, such as PyPDEVS (Van Tendeloo and Vangheluwe, 2014) or DEVS-Ruby (Franceschini et al., 2014), are based on interpreted languages, offering a more readable and intuitive syntax for model developments, at the expense of performances. A hybrid approach consists in implementing parts of the simulator in a compiled language, while offering a more flexible language to develop models. DesignDEVS (Goldstein et al., 2017) lies in this group: mainly implemented in C++, models can be programmed with the Lua scripting language.

With the use of the Crystal language, Quartz lies in the first group of simulators implemented with a technical language. However, due to Crystal similarities with the Ruby syntax, its expressiveness is closest to simulators belonging to the second group of simulators. As such, Quartz is able to produce an optimized executable since Crystal compiler relies on the LLVM (Low Level Virtual Machine) infrastructure while offering a straightforward syntax to program models with.

This paper is organized as follows. We first introduce the theory behind the PDEVS formalism and its capabilities. Afterwards, an overview of the framework is given to show how our concerns impacted its design. We then focus on the features we introduce to facilitate model development. Finally, we present the applications domains that originally motivated the de-

Franceschini, R., Bisgambiglia, P-A., Bisgambiglia, P. and Hill, D.
An Overview of the Quartz Modelling and Simulation Framework.
DOI: 10.5220/0006864201200127
In *Proceedings of 8th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2018)*, pages 120-127
ISBN: 978-989-758-323-0

velopment of Quartz and on which we carry our modelling and simulation efforts.

2 BACKGROUND

The theory of modelling and simulation (TM&S) proposed by (Zeigler et al., 2000) allows describing systems using the Parallel Discrete Event System Specification (PDEVS) in a modular and hierarchical way. A coupled model describes a system as a network of components while an atomic model describes the autonomous behavior of a system. Formally, an atomic model is described by $M = (X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con})$, where X and Y corresponds to the sets of input and output port-values. S is the state of the system. Behavior is defined by several state transitions, triggered as events occur. There are two types of events: one is triggered by the model itself and associated to the internal transition (δ_{int}); the second one depends on inputs of the model and is associated to the external transition (δ_{ext}). A third transition function (δ_{con}) allows to handle simultaneous events of both kinds. Internal events are planned via the time advance function ta. When they occur, the model can produce outputs via the λ function.

PDEVS has well-defined operational semantics thanks to the definition of abstract simulators, which means models can be executed unambiguously on a conforming implementation of the formalism. As (Vangheluwe, 2000) previously showed, its generic properties allows to represent other formalism. Consequently, models expressed in several formalism can be transformed to be executed in the same simulation framework.

3 OVERVIEW

Quartz is a modelling and simulation framework implemented in the Crystal programming language. It can be considered as a replacement of DEVS-Ruby (Franceschini et al., 2014). Crystal syntax is largely inspired if not identical to the Ruby syntax. Mainly, differences between the two lies in the possibility for Crystal to add type annotations, this one being statically typed with type inference.

The architecture is designed to be generic and extensible, to allow the addition of other PDEVS-based formalisms. To help the modeler design its models, we expose an API (Application Programming Interface) in agreement with the terms and concepts used or defined by the theory of M&S defined by (Zeigler et al., 2000). The architecture respects as much as possible the explicit separation principle between modelling and simulation. This way, the modeler can appreciate a familiar architecture, and can easily go from the model specification to its implementation. The generic trait of the software is clearly due to an extensive use of design patterns (Gamma et al., 1994), such as Composite, Visitor, Observer, or Publish-Subscribe. Figure 1 gives an overview of the simulation components at play in the architecture, while Figure 2 shows components at play in the modelling part.

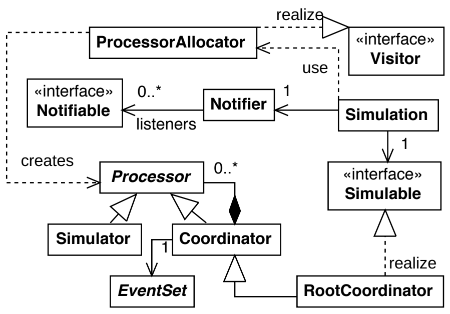

Figure 1: Simplified UML class diagram of simulation components.

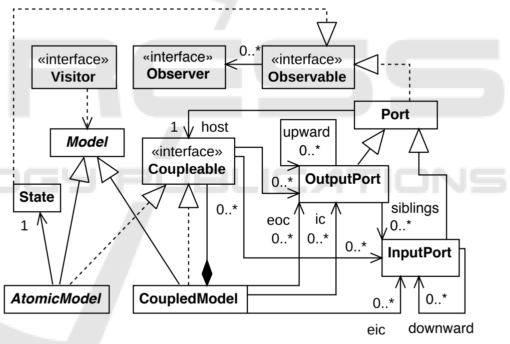

Figure 2: Simplified UML class diagram of modeling components.

To better explain our choices, we now develop each trait that shaped the software design, in particular: the ability to react to events, modularity, and performances.

3.1 Events

There are two types of events issued while simulating a model.

The first type of events is either related to model state changes or to message passing through model couplings. Those are issued on a per model-basis through the Observer pattern. Such design can be used to develop specific pieces of software that can react to models dynamics. For example, one can develop a graphical user interface for its model to help him better visualize the state of a model, which can be laid out when necessary. Based on the same idea, observing a port of a given model may be useful for a piece of software responsible for plotting its values. This pattern establish a simple way to react to the evolution of models. It can be used by the modeler for model-specific code, but can also be used by a larger M&S environment relying on Quartz as a simulation kernel.

The second type of events concerns the simulation itself. They are issued to inform in which state the model execution is. To propagate them, we use the Publish/Subscribe pattern, so that software components may be informed to one or more of the following events, depending on the messages they subscribed for: 1) before/after simulation is initialized; 2) before/after simulation execution; 3) before/after a simulation is abandoned (through an error or a cancellation); 4) before/after a simulation is restarted. Simulation events can be useful at many different levels. For example, the modeler can use them in its models to prepare or cleanup resources before and after a simulation (*e.g.* open or closing a file). For the implementers, this is an opportunity to develop new features while having loose coupling between software components. Finally, simulation events facilitates the integration of the tool in a larger M&S environment. In the same way, external plugins can also take advantage of those events. Also, the exposed simulation API provides control to execute a simulation step-by-step, to cancel it, to restart it, or to execute it until a given virtual time.

3.2 Modularity

Through a modular design, our objective is to make easier introducing new formalisms compatible with PDEVS semantics (*e.g.* a PDEVS extension or a formalism that can be translated to PDEVS). Here, we focus on two aspects illustrating Quartz modularity: decoupling efforts of the I/O interface and tunable event sets.

To support interactions between models, PDEVS defines an input and output interface for each model, so they can be coupled together. However, modular couplings through I/O ports is not the only way to allow models to interact. In order to support formalisms based on both modular and non-modular interactions, a decomposition of the model is necessary to extract its capacity to have I/O ports, which Quartz does by providing a specific `Coupleable` interface (see Figure 2).

As (Van Tendeloo and Vangheluwe, 2014) suggests, the most suitable scheduler directly depends on the nature of the model. Therefore, we let the modeller configure the scheduler for a specific simulation usage. For cases where direct connection is disabled, the modeler can configure the scheduler on a per coupled model basis. We provide several event sets, gathered as subclasses of the `EventSet` abstract class (see Figure 1). To obtain better performances we implemented two multi-list based event sets, which as their names suggest rely on multiple lists to handle events, deferring sorting process until really needed. We also provide heap-based event sets, which can be preferred by the modeller for a more predictable operation cost. For cases where a specific event set is needed, the modeler can eventually implements its own `EventSet` subclass.

3.3 Performances

This section addresses our efforts to reduce the processing time between each scheduled event. We first focus on the various optimizations we implements, based on state-of-the art techniques. Then, through an experiment, we show Quartz provides relative good performances in comparison with other similar tools.

3.3.1 Optimization Techniques

A substantial amount of work dedicated to performance improvement can be found in the literature. As for Quartz, we treated the following aspects to offer correct performances: 1) replacement of the original message passing communication model by sequentializing abstract simulators; 2) simplification of the model hierarchy when it has a depth superior to one; 3) implementation of a tunable event set, relying on multiple priority queues where only active models are considered.

Similarly to (Muzy and Nutaro, 2005), (Himmelspach and Uhrmacher, 2006) or (Vicino et al., 2015), Quartz is based on a sequential execution of models. Message passing suggested in the abstract simulators is not required and can equivalently be represented by a simple function call and the use of return values. As pointed out by (Vicino et al., 2015), this call/return mechanism results in a predictable call stack, which is linear to the depth of the model hierarchy.

Note that if an important number of intermediate coupled models are to be simulated, the depth of the hierarchy may cause a performance degradation, if not cause a stack overflow. As a solution, we use the direct connection (Chen and Vangheluwe, 2010), an optimization technique that transforms the model hierarchy in order to simplify it. The goal is to reduce the depth of the tree by removing intermediate coupled models, and by adjusting couplings between models accordingly.

Direct connection allows to shorten the path a message must take, but does not necessarily imply improved execution time since as a result, the number of models to be managed by a now unique coordinator is larger. Depending on the naiveness of the event set associated with the coordinator, which is responsible to schedule events, processing time may become worrying. Several multi-list based event sets, such as the ladder queue (Tang et al., 2005) claim constant amortized complexity for their operations. Those organize events in different groups, which allows to postpone sorting process until really needed, only on a small part of queued events. Quartz provides an implementation of some of these structures, and lets the modeller choose which to use as described in section 3.2.

3.3.2 Experimental Comparison

In order to evaluate performances of the framework in comparison with other tools, results of an experimental benchmark is proposed. We compare Quartz with DEVS-Ruby (Franceschini et al., 2014), the tool which influenced the presented framework, and with aDEVS (Nutaro, 1999) which is considered in the literature as one of the most performing PDEVS based simulator (Van Tendeloo and Vangheluwe, 2016).

The DEVStone benchmark (Wainer et al., 2011) is used, it has been specifically proposed to compare DEVS simulators and generates automatically a model hierarchy following a configurable structure and behavior. The environment used for our comparison is based on a Intel CoreTM i5-3360M, 16 GB (2xDDR3 1600Mhz) of RAM and a Toshiba MK5061GS hard drive. The software stack relies on Ubuntu 16.04, the Quartz framework (commit a663dfe) compiled with Crystal 0.24.0 (in release mode), aDEVS 2.8.1 compiled with gcc 4.8.2 (-O3 flags), and DEVS-Ruby 0.5.1 interpreted by Ruby 2.2. Mean wall clock time of 10 repetitions is measured for each platform. DEVStone is configured with a fixed depth of 3, the ”HI” coupling type and transition functions times set to 0secs. The varying parameter is the width, which corresponds to the number of models in each level of the hierarchy.

Figure 3 is a graph of the results for each benchmarked simulator. As we hoped for, the port of DEVS-Ruby in Crystal results in a substantial performance benefit since Quartz is significantly faster than DEVS-Ruby. Moreover, Quartz is on the same order of performance as aDEVS, which validates its effectiveness.

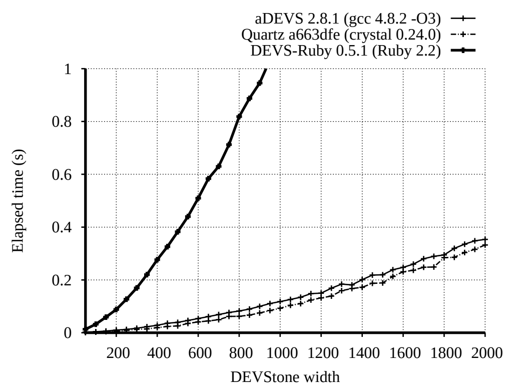

Figure 3: Mean execution times of each platform for the DEVStone benchmark with a varying numbers of models.

4 MODEL DEVELOPMENT

To facilitate model development and its execution, we address two particular issues. Given Quartz is a framework, no GUI is proposed to help model design. However, we propose dedicated expressions which make programming models less tedious while allowing to gather data about models statically. Also, in order to guaranty models respects PDEVS syntax, we work on a verification system that allows to detect errors statically or at runtime.

4.1 Dedicated Expressions

Our framework does not provide a graphical environment for the modeling phase. Instead, we rely on the modeller capability to program its models given the exposed API. Given models must be programmed, an important aspect is to lower the barrier by easing code reading and writing. With meta-programming techniques, we offer a set of dedicated expressions. Crystal supports lexical closures and has a macro system allowing to react to specific changes during the construction of the abstract syntax tree (AST) of a program, which makes it well adapted to the definition of a domain-specific language. Since the main part of model code is found in atomic models, we here focus on specific expressions we offer to ease programming. Those are implemented using macros and are part of the DSEL. The first ones ease ports definition whereas the second one ease state variables declaration.

4.1.1 Ports Declaration

Two expressions allows to declare ports associated with a model at the class-level. Adding an input port is associated with the `input` keyword; adding an output port is associated with the `output` keyword. Both are formally defined by the following BNF expression:

```
<ident> ::= /a-ZA-Z_/{/a-ZA-Z0-9_/ }
<portname> ::= <ident> | ('"' <ident>
 '"') | (':' <portname>)
<portdef> ::= ("input" | "output")
<portname> { ',' <portname> }
```

Using such keywords is more concise than using the provided API. Without them, the modeler has to overload the constructor and use specific methods to attach ports to the the model.

4.1.2 State Variables Declaration

Another expression is dedicated to the declaration of a state variable associated with a model. The modeler is then able to specify the name and type of each variable using the `state_var` keyword, and may also provide an initial value to it. It is defined by the following BNF expressions:

```
<type> ::= /A-Z/ { /a-zA-Z0-9/ }
<svar> ::= "state_var"
<ident> ':' <type> [('=' <arg>) | <
 block>]
```

Note that `<arg>` and `<block>` terms represents any argument that is assignable to a lexical closure as defined by the Crystal syntax. There is no restriction on which type may be used as a state variable, so it is possible to use advanced data structures. In addition to make state variable declaration shorter, this macro allows to identify statically all instance variables that are part of the state of a model. We take advantage of this via meta-programming techniques in order to provide several features seamlessly by generating automatically: 1) model constructors for model initialization; 2) encapsulation of the state variables; 3) a specific `State` subclass associated with the model; 4) the serialization and deserialization of the model state; 5) specific code to ease model parameters exploration.

4.2 Verification

Verification can be used to make sure a model does not include errors. This process can be achieved on different levels: either statically (before or during compilation) or at runtime (during a simulation). Some errors are tied to the specificities of a model, whereas others are more generic. Generic ones can be verified no matter which model is to be verified. As one of our aim is to enforce modeling constraints, Quartz attempts to detect some errors both at runtime or statically.

4.2.1 Runtime Verifications

Quartz applies some generic verifications at runtime. Those are related to generic properties that can be verified for all PDEVS models, and includes: 1) result positivity when applying the time advance function; 2) couplings validity (*e.g.* no feedback loops, involved models belonging to a common coupled model or to the coupled itself); 3) belonging of ports referenced in a given output to the appropriate model; 4) compatibility of the processor with its associated model.

As for errors related to models themselves, a feature allow the modeler to express specific properties about their models that can be checked during simulation. Albeit this feature is not meant to perform an exhaustive exploration of all possible states of a model, invalid states can eventually be detected during an iteration of the M&S process.

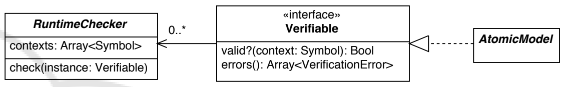
RuntimeChecker
contexts: Array<Symbol>
check(instance: Verifiable)
0..*
«interface»
Verifiable
valid?(context: Symbol): Bool
errors(): Array<VerificationError>
AtomicModel

Figure 4: UML class diagram of components involved in runtime checks.

Figure 4 shows software components used to verify the state of a model. Atomic models implement the `Verifiable` interface, and consequently hold a reference to components encapsulating user-defined rules used to test properties (*i.e.* `RuntimeChecker`). The modeler can specialize this component or use a predefined subclass.

In order to perform such verifications, each new state produced after a transition triggers a check. If the feature is enabled and that an error is detected, a strict mode interrupts the simulation, while a normal model warns the user.

A dedicated expression can be used to define properties to be checked. Following the approach we described in section 4, properties are declared using the `check` keyword at the class-level, in the same way as for the `state_var` keyword.

Listing 1 shows the definition of a sample model. In this example, all instances of `SampleModel` will have a state "bearing" whose type is a 2 bytes integer representing an angular value. The `check` keyword is used to verify that the angle is always in $[0, 360[$.

```
class SampleModel < AtomicModel
  state_var bearing : Int16 = 90
  check :bearing, numericality: {gte:
 0, lt: 360}
  def internal_transition
    @bearing = -123
  end
```

```
end
```

Listing 1: Definition of a sample model using dedicated expressions.

The internal transition assigns an invalid value to the bearing variable. The error is catched at runtime and produce the following output:

```
(10:17:51:273) > 'nav' is invalid (
 context: 'internal_transition', vtime:
  1.0). Errors: ['bearing' must be
 greater than or equal to 0]
```

4.2.2 Static Verifications

Compared to checks performed at runtime, static verification can detect errors much earlier in the development process of a model. It is preferable since compilation can be aborted if errors are detected. According to the adopted metamodel, a language type system already allows to statically raise some structural errors. However, some syntactic errors are more difficult to detect and require a specific analysis of the program syntax tree. Using Crystal macro system, we are able to react to specific events during compilation. For example, the `inherited` macro definition within a class offers an opportunity to apply verifications to all its subclasses.

As a proof of concept, we give an example by checking if the time advance function of an atomic model is syntactically correct. Since Quartz use instance variables to store the state of an atomic model, any method can access and mutate its state. If this is consistent with PDEVS regarding transitions functions, it is not the case for the time advance function. Here, all methods that belongs to `AtomicModel` subclasses are checked. When the method corresponds to the time advance function, its body is inspected via a macro. If an instance variable marked as a state variable is modified, compilation is aborted with a contextual error message.

```
class MyModel < AtomicModel
  state_var x : Int32
  def time_advance
    @x = 54
    return INFINITY
  end
end
```

Listing 2: Definition of an erroneous atomic model.

An example, Listing 2 defines an atomic model which is valid regarding Crystal language semantics, but invalid regarding PDEVS semantics. This is caught at compile time by the rule described earlier and produces the following output:

```
Error in examples/mymodel.cr:21: "you
 are not supposed to mutate the state
 of an atomic model from the #
 time_advance method."
    @x = 54
    ^~~~~~~
```

We should emphasize this particular check can be avoided by adopting a metamodel which strictly conforms to the formalism. We could instead encapsulate the state in an immutable data type and pass it to the time advance function, at the expense of expressivity.

We defined additional rules using this technique to enforce PDEVS constraints, such as preventing ports creation or removal outside the constructor, preventing state modification from λ function, or preventing the deposit of an output value outside λ.

However, this technique has limits. Crystal macros are interpreted and can slow down compile times. Also, more complicated rules are difficult to implement and hard to follow given the particular syntax of Crystal macros. We currently explore an alternative which consists in developing a static analysis tool to freely explore the abstract syntax tree. Since Crystal compiler is self-bootstrapped, we can use its parser and lexical analyzer to implement our model analysis on top of the semantic analysis done by the compiler itself.

5 APPLICATIONS

Due to its generic design, Quartz can be used to develop any kind of PDEVS models and is not tied to a specific domain. As we showed in section 3, its modularity promotes the integration of new formalisms provided that their semantics conforms to that of PDEVS.

However, two main reasons motivated the development of Quartz. The first one was to support a multi-component approach, where models can influence each other directly instead of interacting through I/O ports. The second reason was to facilitate the modelling and simulation of agent-based models by implementing the Dynamic Parallel Discrete Event Multi-Agent Specification (Franceschini et al., 2017), which formalizes the entities at play in a multi-agent system using PDEVS constructs.

5.1 Multi-component Models

We used Quartz to implement the multiPDEVS formalism (Foures et al., 2018), which enable interactions between components in a non-modular way. Instead of interacting through messages, components influence each other directly via state transitions. While multiPDEVS lies in a lower level than a modular coupled network of systems such as PDEVS, some class of systems may be best modelled in a non-modular way. For example, systems such as cellular automata, individual-based models, and eventually cellular environments in agent-based systems may be good candidates.

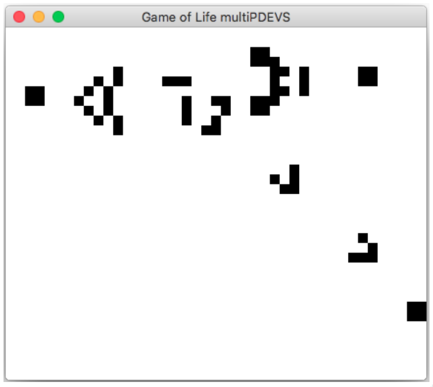

Figure 5: A visualization of a classic Game of Life model based on the multiPDEVS formalism.

Both modular and non-modular approaches can be mixed to represent a particular system. First formally, since a multiPDEVS model can be coupled together with other PDEVS models, as showed in (Foures et al., 2018), and secondly, on the implementation side as showed in section 3.2.

Although it can be used for other purposes, we use multiPDEVS to represent spatially-explicit systems. For example, Figure 5 shows a visualization of a cellular automaton modelled using multicomponent capability of Quartz: the classic John Conway's Game of Life. In a similar way, we used the multi-component approach to represent a firespread model as a cellular automaton (Foures et al., 2018). As we will see in the next section, we also use multicomponent models within modular systems to represent multi-agent systems.

5.2 Agent-based Models

Although Quartz is provided as a generic modelling and simulation framework and as such, can be applied to various domains, we personally intend to apply it for multi-agent based simulation.

This software original main purpose was to provide a simulation kernel for a multi-agent system specification based on PDEVS and various extensions. The DPDEMAS (Dynamic Parallel Discrete Event Multi-Agent System) specification (Franceschini et al., 2017) formalizes generic entities which may compose an agent-based model. Given the amount of components and interactions involved in a multi-agent system and the diversity of application domains, it is difficult to express it in a single formalism. As mentioned in section 2, PDEVS is often used as a pivotal formalism. Models originally defined in other formalisms can be equivalently expressed using PDEVS. Hence, PDEVS was chosen to define DPDEMAS since it allows to couple heterogeneous models modularly. DPDEMAS provides a well-defined structure for components that can be found in agent-based models, such as agents and environments. Semantics are also well defined, since actions performed by agents in their environment or interactions between agents are also formalized.

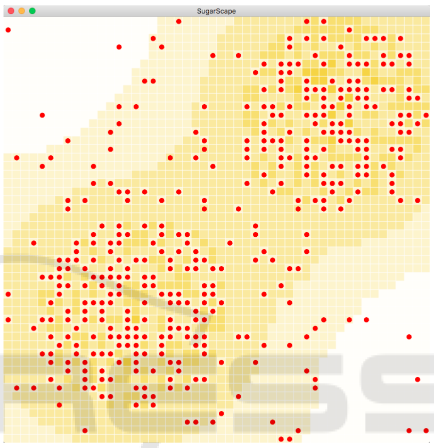

Figure 6: A visualization showing the environment of the SugarScape agent-based model (Epstein and Axtell, 1996).

Following this approach and the Quartz framework, we replicated several classic agent-based models, such as the SugarScape (Epstein and Axtell, 1996) model. Figure 6 shows a visualization of the SugarScape multi-agent system implemented using the DPDEMAS specification.

6 CONCLUSIONS

In this paper, we gave an overview of the Quartz modelling and simulation framework. We described its architecture as well as its features. The modular and extensible architecture conforms with the principle of separation between modelling and simulation. By this mean, integration of new formalisms is possible. The API allows to react to events issued by the models or their processors during simulation. We worked on several aspects of the framework to provide acceptable performances, in particular: (1) a sequential architecture, which provides predictable execution, (2) schedulers providing a constant amortized complexity for all operations and (3), an automatic transformation of the model hierarchy to a depth of one. Originally inherited from DEVS-Ruby (Franceschini et al., 2014), we showed that Quartz allows to combine expressiveness with the performances of a compiled language. To facilitate model development, we described a set of dedicated expressions aiming to simplify model programming, as well as a verification technique to detect potential errors about models, either statically or at runtime.

As a perspective of this work, we wish to develop Quartz by focusing on four aspects: (1) grow the set of dedicated expressions in order to offer a more complete domain-specific language and eventually consider a domain-specific external language; (2) improve static verifications based on macros by developing a static analysis tool; (3) implement distributed simulation algorithms; (4) promotes the tool through a rich documentation to facilitate its use.

ACKNOWLEDGEMENTS

This work is supported by the Corsican region (CTC) and the European Union (EU) through a PO-FEDER regional program: MoonFish.

REFERENCES

Chen, B. and Vangheluwe, H. (2010). Symbolic Flattening of DEVS Models. In *Proceedings of the 2010 Summer Computer Simulation Conference*, pages 209–218, San Diego, CA, USA. Society for Computer Simulation International.

Chow, A. C. H. (1996). Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator. *TRANSACTIONS of the Society for Computer Simulation International*, 13:55–67.

Epstein, J. M. and Axtell, R. L. (1996). *Growing Artificial Societies*. Social Science from the Bottom Up. The MIT Press.

Foures, D., Franceschini, R., Bisgambiglia, P.-A., and Zeigler, B. P. (2018). multiPDEVS: A Parallel Multicomponent System Specification Formalism. *Complexity*, 2018(3751917):19.

Franceschini, R., Bisgambiglia, P.-A., and Bisgambiglia, P. (2017). Approche formelle pour la modélisation et la simulation de systèmes multi-agents. In *Vingt-cinquièmes Journées Francophones sur les Systèmes Multi-Agents*, pages 193–202, Caen, France.

Franceschini, R., Bisgambiglia, P.-A., Bisgambiglia, P., and Hill, D. R. C. (2014). DEVS-Ruby: a Domain Specific Language for DEVS Modeling and Simulation (WIP). In *DEVS 14: Proceedings of the Symposium on Theory of M&S*, pages 393–398. SCS International.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Sofware*. Addison Wesley.

Goldstein, R., Breslav, S., and Khan, A. (2017). Practical aspects of the DesignDEVS simulation environment. *SIMULATION*.

Himmelspach, J. and Uhrmacher, A. M. (2006). Sequential Processing of PDEVS models. In *Proceedings of the 3rd EMSS*, pages 239–244.

Muzy, A. and Nutaro, J. (2005). Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. In *2008 12th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT)*, pages 273–279. IEEE.

Nutaro, J. (1999). ADEVS (A Discrete EVent System simulator). *Arizona Center for Integrative Modeling {&} Simulation (ACIMS), University of Arizona, Tucson. Available at http://www.ece.arizona.edu/nutaro/index.php*.

Quesnel, G., Duboz, R., and Ramat, E. (2009). The Virtual Laboratory Environment – An operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Simulation Modelling Practice and Theory*, 17(4):641–653.

Tang, W. T., Goh, R. S. M., and Thng, I. L.-J. (2005). Ladder Queue: An O(1) Priority Queue Structure for Large-scale Discrete Event Simulation. *ACM Transactions on Modeling and Computer Simulation*, 15(3):175–204.

Van Tendeloo, Y. and Vangheluwe, H. (2014). The Modular Architecture of the Python(P)DEVS Simulation Kernel Work In Progress paper. In *Proceedings of the Symposium On Theory of Modeling and Simulation (TMS'14) SpringSim 2014*, pages 387–392, Tampa, FL, USA. SCS.

Van Tendeloo, Y. and Vangheluwe, H. (2016). An evaluation of DEVS simulation tools. *SIMULATION*.

Vangheluwe, H. (2000). DEVS as a common denominator for multi-formalism hybrid systems modelling. In *IEEE International Symposium on Computer-Aided Control System Design, 2000. CACSD 2000*, pages 129–134. IEEE.

Vicino, D., Niyonkuru, D., Wainer, G. A., and Dalle, O. (2015). Sequential PDEVS Architecture. In *DEVS 15: Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 906–913, Alexandria, VA, USA.

Wainer, G. A., Glinsky, E., and Gutierrez-Alcaraz, M. (2011). Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. *SIMULATION*, 87(7):555–580.

Zeigler, B. P., Kim, T. G., and Praehofer, H. (2000). *Theory of Modeling and Simulation, 2nd Ed.* Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press, Orlando, FL, USA, 2nd edition.