Sequential PDEVS Architecture

Damián Vicino[1, 2] **Daniella Niyonkuru**[1], **Gabriel Wainer**[1] **Olivier Dalle**[2, 3]
[1]Dept. of Systems and Computer Engineering
Carleton University 1125 Colonel By Dr., Ottawa, ON, Canada K1S 5B6
[2]Université Nice, I3S, UMR CNRS 7271, Sophia Antipolis, France [3]INRIA Sophia Antipolis, France
{Damian.Vicino,Daniella.Niyonkuru,Gabriel.Wainer}@carleton.ca olivier.dalle@unice.fr

ABSTRACT
Parallel Discrete Event System Specification (PDEVS) is a well-known formalism used to model and simulate Discrete Event Systems. This formalism uses an abstract simulator that defines a set of abstract algorithms that are parallel by nature. To implement simulators using these abstract algorithms, several architectures were proposed. Most of these architectures follow distributed approaches that may not be appropriate for single core processors or microcontrollers. In order to reuse efficiently PDEVS models in this type of systems, we define a new architecture that provides a single threaded execution by passing messages in a call/return fashion to simplify the execution time analysis.

Author Keywords
PDEVS; sequential; architecture; simulator;

ACM Classification Keywords
I.6.8 [SIMULATION AND MODELING]: Types of Simulation---Discrete event; D.1.4 [PROGRAMMING TECHNIQUES]: Sequential Programming; D.2.11 [SOFTWARE ENGINEERING]: Sequential Programming

INTRODUCTION
Discrete Event System Specification (DEVS) is a mathematical formalism for modeling and simulating discrete-event dynamic systems [1]. To tackle the complexity of the system, DEVS decomposes the system into basic (behavioral) models called atomic models and composite (structural) models called coupled models.

In classic DEVS, whenever two models are scheduled for state transitions at the same time, one of the models is chosen according to a *select* function provided in the coupled model specification. Classic DEVS was extended to resolve serialization constraints and allow simultaneous events, defining what is called Parallel DEVS (PDEVS) [2].

Each DEVS specification can be executed by an abstract simulator that defines the operational semantics of the models. Existing PDEVS simulation algorithms are parallel by nature, and lead most developers to adopt a concurrent implementation (even in single cores). Nevertheless, a sequential algorithm could lead to better performance while dealing appropriately with simultaneous events and guaranteeing predictability. Moreover, on low-cost single core processors, concurrent implementations will be executed sequentially by the operating system, and this may introduce unpredictability and overhead. In critical systems for instance, reliability and predictability are more important than expressiveness and performance. In these cases, it is essential to produce a program with a predictable execution trace.

Here, we show new sequential algorithms to run PDEVS models, using a modular and flexible architecture. The objective is to achieve high performance and provide predictability. The algorithms have a small call stack, linear to the height of the model hierarchy. This aspect is particularly essential for the previously discussed systems that may have limited resources. To demonstrate the quality of the proposal, the simulator is compared against other simulators in the context of single core execution.

A simulator that implements the above concepts was developed and follows the C++11 standard. Therefore, its API can be integrated to Boost, a popular set of libraries for C++, providing a library of discrete-event system specifications using DEVS for the community of C++ standard programmers.

BACKGROUND
Our proposed approach is based on the abstract algorithms defined by Chow [2]. The simulation algorithms can be found in [2]. Chow also implemented a set of algorithms for distributed computation in [3].

To the best of our knowledge, the only sequential approach that also preserves the coordinator/simulator concepts (as defined in the abstract simulator) is JAMES II [4]. This approach, which is partially sequential, was compared against parallel versions showing that it performed better for larger models, and was less efficient for smaller ones. The tests used an example of forest fire simulation.

Several other simulators have implemented PDEVS using different approaches. We will first discuss *adevs* (a Discrete Event Simulator) since in [5] it was the fastest of the DEVS framework tested. In *adevs*, coupled models (also referred to as network models in [6]) are reduced to an equivalent atomic model (called the resultant) whose states, transition and output functions are defined by its interconnected components. *Adevs* exploits the closure property and converts coupled models into atomic models with corresponding transition, output and time advance functions [7] through the resultant transformation, and hence eliminates the need of simulating each component individually.

Adevs has evolved during the last 15 years, has over 29 releases, and supports high performance by relying on optimized data structures. For instance, the Set implementation used in *adevs* is a dynamic array backed by a hash table for retrieving specific items. The scheduler is based on an array-based binary heap and permits fast rescheduling [8].

PyPDEVS [9] was recently implemented and its per-

TMS/DEVS 2015, April 12 - 15, 2015, Alexandria, VA, USA

formance now comes close to *adevs* [5][9]. PyPDEVS offers a modular architecture, using a *BaseSimulator* class to run both coupled and atomic DEVS models as it applies symbolic flattening [9][10]. Besides, it offers a variety of schedulers (namely a sorted list, activity map, and a heapset) for speedup. Unlike *adevs*, PyPDEVS supports dynamic typing; therefore, all the messages are not required to be of the same type. PyPDEVS has sequential and distributed variants [11] but both mainly focus on computational activity information in order to reduce simulation time.

Other PDEVS simulators include VLE (Virtual Laboratory Environment) [12] that couples multiple simulators within a DEVS-Bus architecture and uses PDEVS for coordination; DEVS-Ruby [13], which uses processors to wrap models, and comes close to PyPDEVS performance [5]; and CD++ [14], whose performance is slower than *adevs*.

The DEVStone synthetic benchmark [15] has been used to evaluate the performance of different DEVS simulators. DEVStone generates a suite of models of different size, complexity and behavior to mimic applications in the real world. DEVS supports the construction of hierarchical models, which may affect performance. DEVStone was created to study and compare the efficiency of DEVS (and other discrete event) simulators, to compare different versions of a specific simulation engine, and to help measuring and improving DEVS-based software.

The DEVStone model generator allows one focusing on essential aspects that impact performance: the size of the model and the workload of its transition functions. The following parameters are used to generate a model: type (structure and interconnections between components), depth (number of levels in the hierarchy), width (number of components in a coupled model), internal transition time and external transition time. There are four types of models available (LI, HI, HO and HOMod), each with a different internal and external structure

- LI: these models have a low level of interconnections, i.e. one input and one output port, for each coupled model. The input port is connected to each component but only one component produces an output through the output port.
- HI: these models have a high level of input couplings. Each atomic component *a* connects its output port to the input port of the $a+1$th component.
- HO and HOmod [15]: these are models with high level of coupling and numerous outputs. HO models have two inputs and two outputs at each level while HOmod have a second set of (width-1) models where each one of the atomic components triggers the entire first set of (width-1) atomic models.

As the model structure and the time spent in transition functions are known (as the transition functions execute the Dhrystone benchmark), the model execution time can be computed. Detailed formulas can be found in [15].

Several simulators have been compared using DEVStone. In [15], *adevs* outperformed CD++ by a significant margin for large models. Several other PDEVS simulators (JAMES II, VLE, PyDEVS, DEVS-Ruby) have been compared to *adevs*, in [5], using DEVStone as well and *adevs* remains the reference with regards to performance. Flattening the model [16] is a proposed improvement in which the model structure is modified so that there are no more coupled models. Hence, the overhead induced by passing messages through different levels disappears. The impact of flattening has also been measured using DEVStone [15], showing clear performance improvement.

Existing simulators have used various mechanisms to parallelize the code, but this often results in high overhead. Likewise, implementing the abstract simulators efficiently and choosing good data structures methodically are essential for the simulator's performance. To tackle these challenges, we introduce a sequential architecture, and we provide an effective implementation of the abstract simulator. The performance is assessed by comparing it to *adevs*.

A SEQUENTIAL ALGORITHM

In the background section, we briefly mentioned the PDEVS abstract simulation algorithms, which uses two kinds of components: Simulators (in charge of Atomic models), and Coordinators (in charge of Coupled models), using a one to one mapping between models and simulation components.

The approach used in the abstract algorithms' definition is parallel: all the components are considered to be running and waiting for a message at the start. The Root Coordinator starts the simulation by sending an *advance* request to all its children and waits until a response is received from all of them. Each Coordinator receiving one of these messages does the same: it sends the message to each of its children and waits for the replies (*done* and *output* messages). When a request reaches a Simulator, it runs sequentially the simulation of its Atomic model and sends results to its parent Coordinator. Once all the replies are collected at the Root level, the Root Coordinator sends the messages to advance the simulation once again, until the end.

Coordinator Algorithms

To provide a sequential, single threaded Coordinator, we removed semaphores and simultaneous executions defined in the original abstract simulator. We also replaced the top-down message passing by function calls, and the bottom-up message passing by return from the called functions. An additional benefit of this approach is that children are not aware of their parent coordinators and do not need to keep track of their parents. Likewise, we are sure that the call hierarchy is limited to a fixed value that is linear to the height of our model hierarchy, since calls are only initiated by the parent coordinators, and they only happen in a top down fashion.

Because multiple message types are supported, replies can carry more than the confirmation of execution. We have encoded the returned message type so that different kind of messages – *done* and *output* messages – can be received from the children. Afterwards, we use the message type to process the reply and to route messages accordingly. For instance, if a child sends an output message, the parent will add this message to the *inbox* of all other children that are internally coupled to the one that generated output.

For the messages received from the parent, we define two functions: *advance-simulation* and *collect-outputs* as shown in Figure 1.

```
Coordinator
Vars:
    next  // Next scheduled event time
    last  // Last processed event time
    FEL   // Future event list

Method collect_outputs(Time t)
   if  t != next then return {}
   else
      set outputs = empty bag
      for each imminent submodel Coordinator c
         if c is in EOC
         then outputs = Union(outputs,
               c.collect_outputs(t) )
         end if
      end for
      return outputs
   end if
end method

Method advance_simulation(Time t)
   assert t in [last, next]
   last = t
   set external_imminents = empty set

   for each Coordinator c of a submodel in EIC
     if self.inbox is not empty and c.next != t
     then add r to external_imminents
     end if
     add self.inbox contents to c.inbox
   end for

   if t == next then
      for each Coordinator c of a submodel receiv-
ing input from an imminent i because of IC
          set temp = collect_outputs(i)
          if not empty temp and c.next != t
          then add c to external_imminents
          end if
          add temp to c.inbox
       end for
   end if

   for each Coordinator c in Union(imminents,
            external_imminents)
       c.advance_simulation(t)
       if c.next != infinity
       then
           FEL.remove_value(c) //for rescheduling
           FEL.insert(c.next, c)
       end if
   end for

   if empty FEL then  next = infinity
   else next = FEL.top.first end if

   imminents = coordinators on top of FEL
   remove imminents from FEL
end method
```

Figure 1. Coordinator

We added a structure to each Coordinator called *inbox*, which is used to collect the messages returned by *collect-output* that will be used by the next call to *advance-simulation*. This is safe since we know that the two functions will always be called in the same order because of how the main loop is defined by the Root Coordinator.

In *collect_outputs*, the coordinator verifies if it has reached its next state change time. If not, an empty reply is sent; otherwise, the outputs of each imminent coordinator of its submodels are collected and added to the output bag. Hence, all the Y-messages are collected first and then sent together.

For *advance_simulation*, the time *t* is verified to ensure that it is between the last and next scheduled change. If so, *t* is saved as the last change time, and external imminent models (those that received an input event) are set by adding each receiver of the external coupling set to the external imminent set, and adding the content of the inbox to the receiver's inbox. The previous steps run if the coordinator inbox is not empty (an input message was received) and the receiver's next state change is not *t*. If it is time for the next state change (*t == next*), the outputs of each imminent model are collected and carried out to any linked coupled model that is then added to the external imminent set.

In all these cases, the coordinator calls *advance_ simulation* for each coordinator in the *imminent* and *external_ imminent* sets, and their next state change time is added to the Future Event List (FEL). If this is empty, the next state change is infinity; otherwise it is picked from the FEL. Finally, all the *imminents* are retrieved from the FEL.

Simulator Algorithms

For the Simulator, only the return reply mechanism is required. Since Coordinators and Simulators do not know their parents, all communications are initiated in a top down fashion, and the replies are collected using the method returned values. The function names are the same as in the Coordinator: *advance_simulation* and *collect_outputs.* The algorithms for the Simulator are shown in Figure 2.

```
Simulator : subclass of Coordinator
Vars:
    next  // Next scheduled event time
    last  // Last processed event time
    model // atomic model being simulated

Method collect_outputs(Time t)
   if  t != next then return {}
   else return model.out() end if
end method

Method advance_simulation(Time t)
   assert t in [last,next]
   if self.inbox is empty and t == self.next
   then
       model.internal()
       next = last + model.time_advance()
   end if

   if self.inbox is not empty
   then
```

```
        set local_t = t - last
        if t == next
        then model.confluence(inbox, local_t)
        else model.external(inbox, local_t)
        end if
        next = last + model.time_advance()
    end if
    last = t
end method
```

Figure 2. Simulator

The *collect_outputs* method verifies the parameter time *t*. If this is different from the next scheduled event, an empty bag is returned; otherwise, the output generated by the model is returned. For *advance_simulation*, we verify if time *t* is legitimate by making sure it is within the last change and the next expected change. If *advance_ simulation* was called with a valid time *t*, the *inbox* content is checked. If the inbox is empty and it is time for the next event, i.e. the next internal transition, the internal function is executed and the next change is set by adding the last change time and the delay TA. When *inbox* is not empty, (an input has been received), we execute the external function if the time different from the next state change (internal transition time). If not, it indicates that the external and internal transitions are scheduled for the same time and consequently the confluent function is executed.

SIMULATOR ARCHITECTURE

The architecture of the new simulator consists of modules that mainly correspond to the modeling and simulation execution engines. This modular design has the great advantage of providing simple interfaces to the modeler, who can easily transition from the PDEVS specification to the model implementation. The simulation mechanism and the abstract simulator details are hidden, and they do not need to be manipulated by the users, but could be extended by an expert user. The architecture components can be grouped into three categories: *Model* classes, *Execution* classes and *Utility* classes as shown in Figure 3.

The *Model* classes are used to build PDEVS models. They include the *PDEVSAtomic* and *PDEVSCoupled* classes that inherit from a class named *Model*. *PDEVSAtomic* provides virtual methods for defining PDEVS atomic models behavior. On the other hand, *PDEVSCoupled* is for organizing the models in a hierarchical manner and coupling components.

The *Execution* classes provide simulation engines for the atomic and coupled models defined by the users. The *PDEVSSimulator* class executes the simulator's algorithms defined in the previous section, and it renders atomic models behavior. The *PDEVSCoordinator* class runs the algorithms described in the sequential algorithm in order to execute coupled models.

The *Utility* classes are not explicitly defined in the PDEVS formalism or the abstract simulator, but they have been included as they provide useful functionalities. These classes are used to control the input stream that process input event files. Other useful components are the *Message*, *Time* and *FEL* (Future Event List) classes.

Model classes Implementation

For model classes (shown in Figure 4), we will first present the *PDEVSAtomic* class, which is used to define new atomic models. The constructor requires two template parameters: Time and Message. The functions provided by *PDEVSAtomic* correspond to those described in the formalism: internal, external, confluent, time-advance and output functions. The time-advance, which is commonly included in the internal and external function in various simulation implementations, is here clearly separated and has its own dedicated function.

Coupled models are defined using the *PDEVSCoupled* class. This class constructor receives four parameters: the list of pointers of the components to be coupled; the External Input Couplings (EIC) pointers list for models that receive inputs from outside of the coupled model; the Internal Couplings (IC) pointers list for models that are connected internally; and finally the External Output Couplings (EOC) pointers list for models that send outputs outside of the coupled model. *PDEVSCoupled* provide two implementations of this constructor: one that takes initialization lists and another with vectors. This is particularly useful for dynamical model construction and certain compilers with unimplemented initialization lists.

Both *PDEVSAtomic* and *PDEVSCoupled* classes inherit the model class that allows coupled and atomic models to be connected easily through couplings that can be debugged with ease since they share a common model interface.

Execution classes implementation

Execution classes, illustrated in Figure 5, implement the abstract simulator algorithms and execute models. The *PDEVSCoordinator* class, in charge of managing coupled models, requires three template parameters: *Time*, *Message* and *Future Event List (FEL)*. These three parameters will be detailed in the *utility* classes.

Constructing coordinator objects is complex, as it requires the coupled model components to be extracted and embedded in the coordinator. For instance, when the coordinator is built, all the children are constructed, and the couplings between components that communicate are saved. The simulation algorithms described previously– *collect_outputs* and *advance_simulation* - are implemented in this class.

The *PDEVSSimulator* class implements the simulator's algorithms presented in the sequential algorithm. Therefore, this class is in charge of calling the state transition functions at the appropriate times, and of returning the outputs of the atomic models to their coordinators.

In addition to the described function, an *init* function setups the model initial state. In addition, an extra variable keeps a pointer to the simulated model.

Apart from the PDEVSSimulator and PDEVSCoordinator classes, a *PDEVSRunner* class was implemented. This class advances the simulation for the Root coordinator and defines the end time of the simulation. It also provides mechanisms for output and debugging.

Utility classes

The *utility* classes provide essential data structures to run the simulation properly. The first class in the utility category is called *Message*. Boost::any is used by default as the message type, and it allows the exchange of any type of messages in our models. This data type enables us to define type agreements between any pair of connected models without restricting the user to utilize a single data type for every communication in the simulation.

For the *time* component, we have developed some complex data types based on other lines of work we are studying at present [17]. In the current implementation, any type having assignation, equality, order, addition and definition of infinite can be used, i.e. double is accepted.

The Future Event List (FEL) is also provided as part of the utility classes. Using an effective FEL is essential in order to achieve good performance, as exemplified by the schedulers implemented in PyPDEVS. For the FEL type, any structure that matches the priority queue signature is accepted. Hence, the user can define customized schedulers and increase performance if needed. The default provided FEL is the standard priority queue for now. This data structure is part of the language and is suited to store and retrieve timed events. Other data structures have been considered for the FEL [18] and will be implemented in the future.

In addition to the data types provided by the above-described classes, an input stream model is also provided. Its role is to allow reading and processing events that originate from an external source. Indeed, in most of the cases, models receive inputs that come from the external environment. A common approach used by many simulators is to add file reading capabilities to the Root coordinator. However, we believe that having a specialized module that assumes this responsibility is better. Indeed, having the event file processed in the root coordinator requires all inputs to be centralized in the same location and involves additional routing. In contrast, a separate input stream model is more flexible and allows the inputs to be placed close to the model of interest. Therefore, the new simulator provides a standard atomic model capable of processing standard input stream. This model receives two constructor parameters: the input stream to be read and the function to process events in the stream.

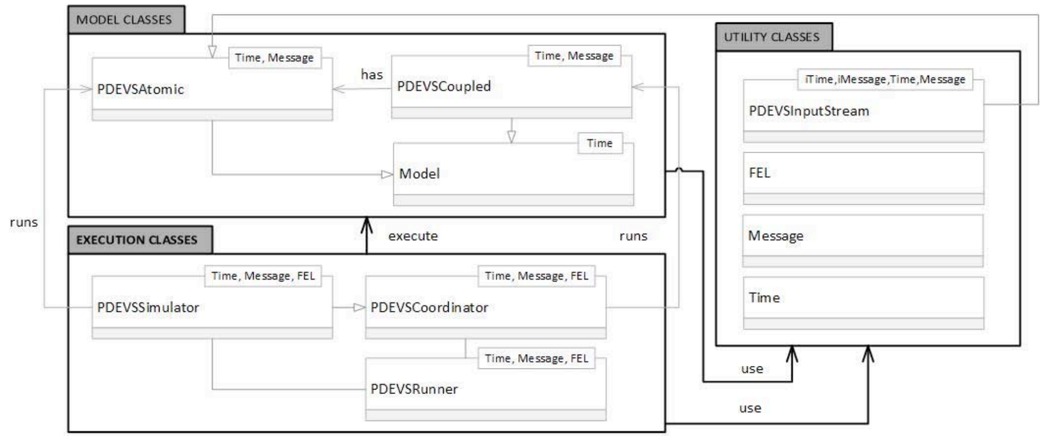
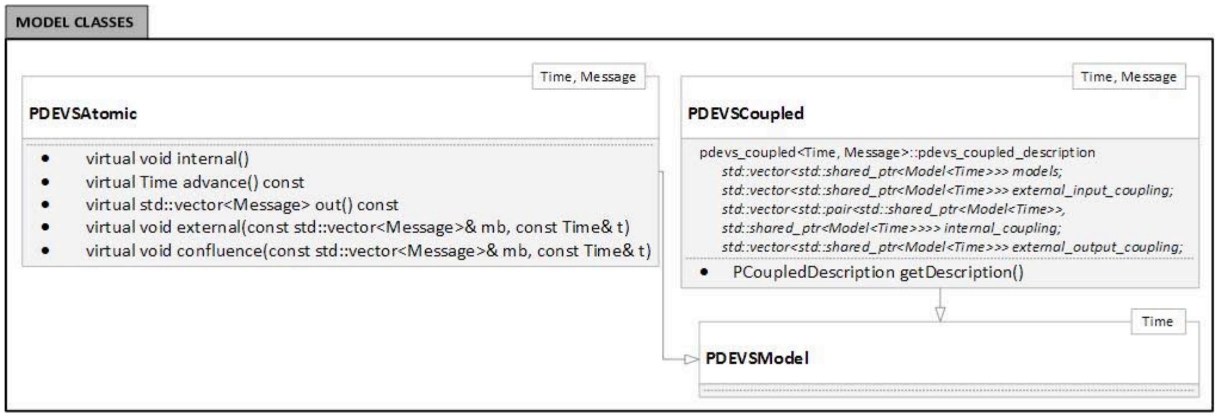


Figure 3. Architecture



```
• virtual void internal()
• virtual Time advance() const
• virtual std::vector<Message> out() const
• virtual void external(const std::vector<Message>& mb, const Time& t)
• virtual void confluence(const std::vector<Message>& mb, const Time& t)
```

```
pdevs_coupled<Time, Message>::pdevs_coupled_description
    std::vector<std::shared_ptr<Model<Time>>> models;
    std::vector<std::shared_ptr<Model<Time>>> external_input_coupling;
    std::vector<std::pair<std::shared_ptr<Model<Time>>,
    std::shared_ptr<Model<Time>>>> internal_coupling;
    std::vector<std::shared_ptr<Model<Time>>> external_output_coupling;
• PCoupledDescription getDescription()
```

Figure 4. Model Classes

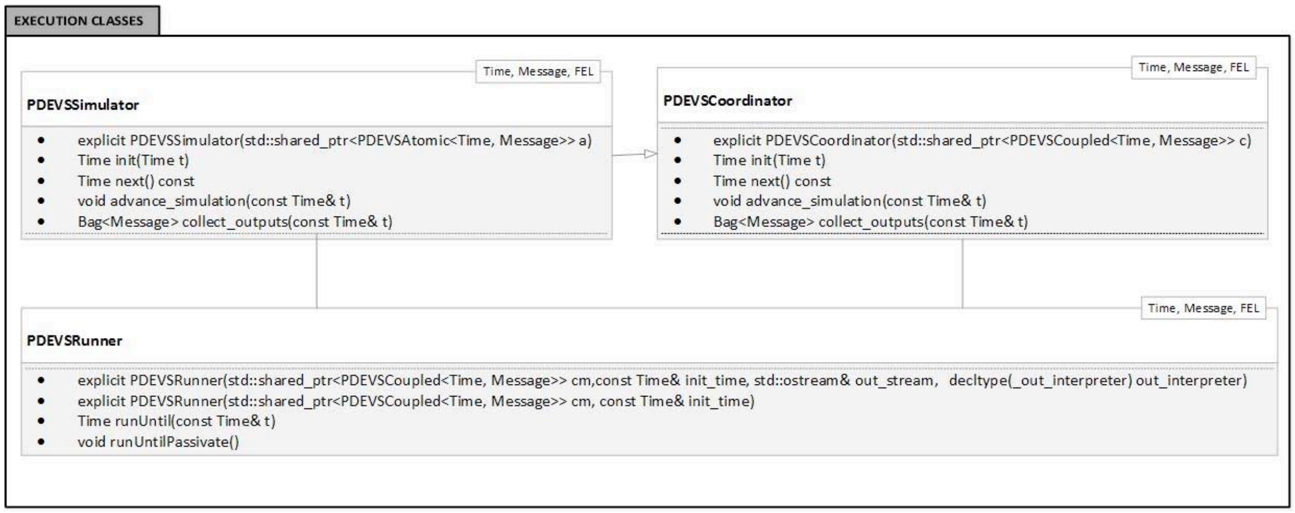

EXECUTION CLASSES
PDEVSSimulator
Time, Message, FEL
explicit PDEVSSimulator(std::shared_ptr<PDEVSAtomic<Time, Message>> a)
Time init(Time t)
Time next() const
void advance_simulation(const Time& t)
Bag<Message> collect_outputs(const Time& t)
PDEVSCoordinator
Time, Message, FEL
explicit PDEVSCoordinator(std::shared_ptr<PDEVSCoupled<Time, Message>> c)
Time init(Time t)
Time next() const
void advance_simulation(const Time& t)
Bag<Message> collect_outputs(const Time& t)
PDEVSRunner
Time, Message, FEL
explicit PDEVSRunner(std::shared_ptr<PDEVSCoupled<Time, Message>> cm,const Time& init_time, std::ostream& out_stream, decltype(_out_interpreter) out_interpreter)
explicit PDEVSRunner(std::shared_ptr<PDEVSCoupled<Time, Message>> cm, const Time& init_time)
Time runUntil(const Time& t)
void runUntilPassivate()

Figure 5. Execution Classes

Finally, it is essential to remind that we plan to submit this simulation code to the Boost Library [19]. Therefore, the Boost coding standards were used and only Boost and the C++11 standard libraries were included. We compiled and tested our code on multiple platforms, including various distributions of Linux, DragonFlyBSD, FreeBSD, MS Windows, OSX, and on an ARM platform (using STM32F2 Evaluation Board). Tool chains available in each platform were used, namely clang, gcc and MS compilers.

In the next section, we present comparisons with *adevs* to assess the new simulator performance.

RESULTS

In order to evaluate the performance of the new simulator, we have designed a set of experiments using the DEVStone benchmark previously described in the background section. These experiments run different tests on both HI and LI models.

The same experiments are also run using the flattened model version of the new simulator and the results are compared to *adevs* for each case. In the following graphs, we will identify the simulators as *adevs*, cdevs and cdflat where cdevs and cdflat both refers to the new simulator, the first one running the model as defined and the second running a flattened version of it.

For our first experiment, we used a LI topology with height 5 and width 1000. We introduced to the simulation sets of input messages, one message per simulation second.

Figure 6 shows the execution time for running this simulation in each of the simulators. Figure 7, shows the results for the same experiment when using HI topology. In this case, we had to reduce the width to 100 due to hardware limitations. In both cases, *adevs* and cdevs/cdflat performances are close – the difference is never higher than 1%. From this experiment, we can conclude that the new simulator has a performance similar to *adevs* for sequential events.

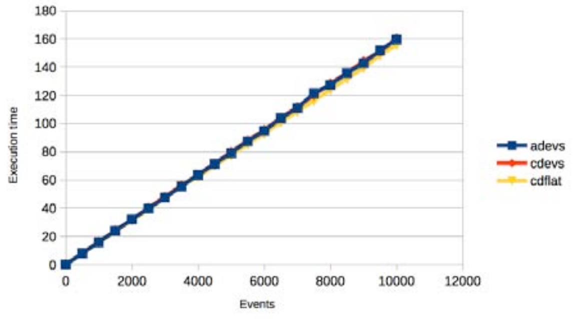


Figure 6. LI with Width = 1000, Height = 5, Individual Events

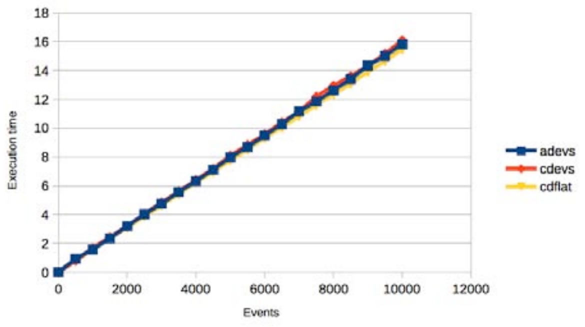


Figure 7. HI with Width = 100, Height = 5, Individual Events

For the next experiment, we used the same models but introduced all input messages simultaneously. This is mainly done in order to verify if our implementation handles simultaneous events as well as *adevs*, which uses the closure property. We show the results for LI topology in Figure 8 and those for HI topology in Figure 9.

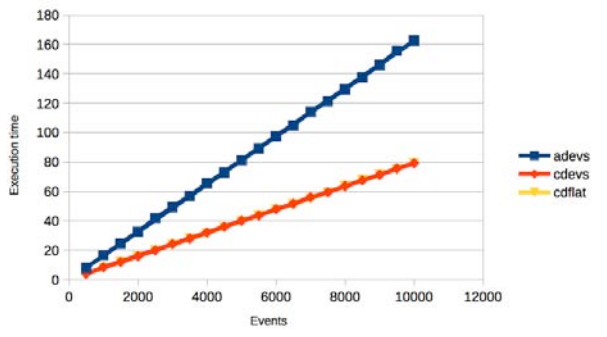


Figure 8. LI with Width = 1000, Height = 5, Simultaneous Events

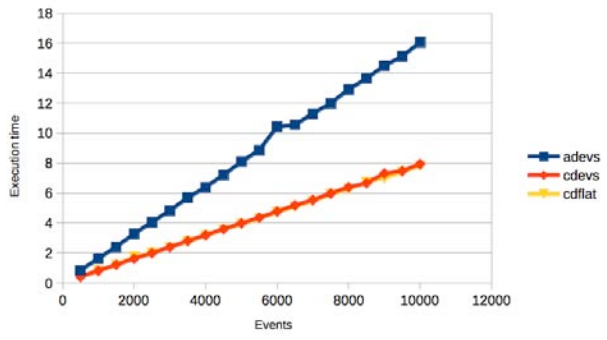


Figure 9. HI with Width = 100, Height = 5, Simultaneous Events

In both cases, the new simulator clearly performs better and handles simultaneous input messages faster than *adevs* as shown in figure 8. Moreover, the difference increases linearly with the number of injected messages.

For the next experiment, we used an LI topology model with different depths and introduced to it 5000 events. In Figure 10, we show the execution time when the messages are introduced sequentially separated by 1 second. In Figure 11, we show execution time when all messages are introduced simultaneously at time 0. We observe a similar pattern as in the previous plots. As illustrated, injecting simultaneous messages produces a difference linear to the depth of the model.

Finally, for our last experiment we use the same LI topology as in the first experiment. We input 10000 messages to the model. The messages are first delivered 1 at a time, then 2 at a time, 5 at a time, until 500 is reached. Figure 12 shows the execution time for these experiments.

Notice that the first few increases in the packet of events size reduce the execution time logarithmically, but when the packages are larger than twenty, performance hardly improves, and stays constant.

This experiment shows that the difference is exhibited when simultaneous events are involved and is proportional to the depth of the model. This can be explained by the fact that *adevs* has to construct atomic models equivalent to coupled models as per the closure property.

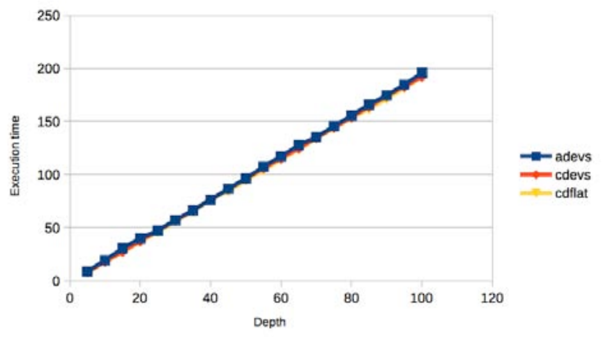


Figure 10. LI with Width = 100, Input Events = 5000, Sequential

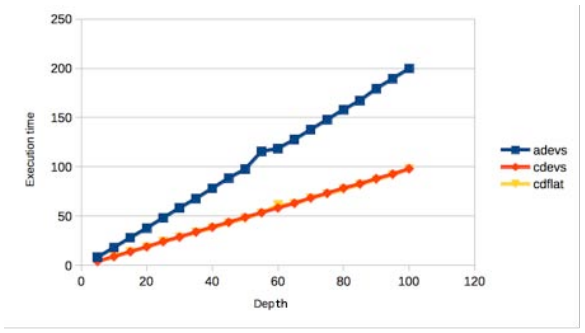


Figure 11. LI with Width = 100, Input Events = 5000, Simultaneous

From the above experiments, we have observed similar results for HI and LI models in the presence of sequential events. When simultaneous events are introduced, the new simulator has an obvious advantage that increases when the number of input messages is increased. For LI models, the difference also increases proportionally to the width.

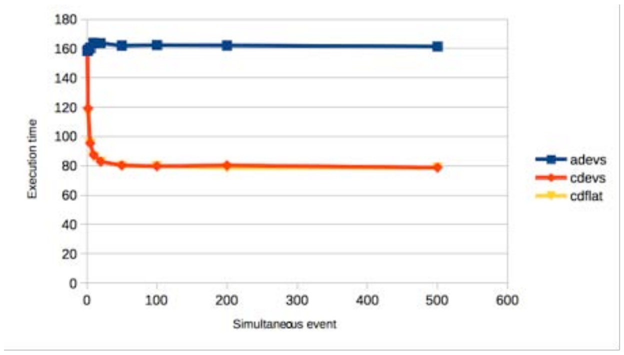


Figure 12. LI Coupling receiving packs of messages

Hence, the new simulator provides an elegant modular architecture that preserves the coupled/atomic structure and allows having a best-in-class performance.

CONCLUSION

The PDEVS formalism was introduced to resolve classic DEVS serialization constraints and allow simultaneous events. In this paper, we presented a PDEVS simulator that implements Chow's abstract simulator. This abstract simulator defines algorithms that are parallel by nature. Yet, a parallel implementation although intuitive might not be the best approach for single core processors. We have used a sequential approach that enables determinism in a simple and neat way, saves resources and is time efficient. This approach is particularly useful for time-critical systems where predictability is of the utmost importance.

A modular architecture that maintains the coordinators/coupled models and simulators/atomic models was also described. This design preserves the natural structure of DEVS models. Furthermore, the simulator uses a set of efficient algorithms for simulators and coordinators and an implementation based on call/return that allows sequential execution.

The new simulator was implemented as a library written in C++, compliant with the C++11 and the Boost library coding standard. It supports multiple data types for the Time, FEL and Messages, and compiles in multiple platforms, including BSD, Linux, OS X, MS Windows and embedded systems.

We compared the new simulator performance against *adevs* using the DEVStone benchmark and showed that no new overheads were introduced. In particular, the new simulator performs better than the *adevs* in presence of simultaneous events, main purpose for which PDEVS was created.

Future work will consist in submitting the library for review to Boost, introducing the concept of ports in the simulator and optimizing the data structures used for the bag data type that carries messages.

REFERENCES

1. B. P. Zeigler, H. Praehofer and T. G. Kim, Theory of modeling and simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, San Diego, CA: Academic Press, 2000.
2. A. C. Chow, B. P. Zeigler and D. H. Kim, "Abstract simulator for the parallel DEVS formalism". *Proc. of the Fifth Conference on AI, Simulation, and Planning in High Autonomy Systems,* Gainesville, FL, 1994.
3. A. C. Chow, "Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator," *TRANSACTIONS of the Society for Computer Simulation,* vol. 13, no. 2, pp. 55-68, 1996.
4. J. Himmelspach and A. M. Uhrmacher, "Sequential processing of PDEVS models," in *Proceedings of the 3rd EMSS*, Barcelona, Spain, 2006.
5. R. Franceschini, P.-A. Bisgambiglia, L. Touraille, P. Bisgambiglia, D. Hill, R. Neykova and N. Ng "A survey of modelling and simulation software frameworks using Discrete Event System Specification," in *2014 Imperial College Computing Student Workshop,* London, England, 2014.
6. J. Nutaro. (2014, Feb 19). *A Discrete EVent system Simulator.* [Online]. Available: http://web.ornl.gov/~1qn/adevs/adevs-docs/manual.pdf
7. J. J. Nutaro, Building software for simulation: theory and algorithms, with applications in C++. Hoboken, NJ: Wiley. 2011.
8. A. Muzy and J. J. Nutaro, "Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulator" *1st Open International Conference on Modeling & Simulation (OICMS)*, Clermont-Ferrand, France, 2005.
9. Y. Van Tendeloo and H. Vangheluwe, "The modular architecture of the python (P)DEVS simulation kernel" *Proceedings of the Symposium on Theory of Modeling & Simulation*, 2014.
10. B. Chen and H. Vangheluwe, "Symbolic flattening of devs models," in *Proceedings of the 2010 Summer Computer Simulation Conference*, Ottawa, Canada, 2010.
11. Y. Van Tendeloo and H. Vangheluwe, "Activity in PythonPDEVS," *ITM Web of Conferences,* vol. 3, p. 01002, 2014.
12. G. Quesnel, R. Duboz, E. Ramat and M. K. Traore, "VLE: a multimodeling and simulation environment," in *Proceedings of the 2007 summer computer simulation conference,* San Diego, CA, 2007.
13. R. Franceschini, P.-A. Bisgambiglia, P. Bisgambiglia and D. Hill, "DEVS-ruby: a domain specific language for DEVS modeling and simulation (WIP)," in *Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative*, Tampa, FL, 2014.
14. G. Wainer, "CD++: A Toolkit to Develop DEVS Models," *Software: Practice and Experience,* vol. 32, no. 13, pp. 1261-1306, 2002.
15. G. Wainer, E. Glinsky and M. Gutierrez-Alcaraz, "Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark," *Simulation,* vol. 87, no. 7, pp. 555-580, 2011.
16. K. Kim, W. Kang, B. Sagong and H. Seo, "Efficient Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-Hierarchical One," in *Proceedings of the 33rd Annual Simulation Symposium,* Washington, DC, 2000.
17. D. Vicino, O. Dalle and G. Wainer, "A Data Type for Discretized Time Representation in DEVS," in *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, Lisbon, Portugal, 2014.
18. J. Himmelspach and A. M. Uhrmacher, "The event queue problem and PDevs," in *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2,* Norfolk, VA, 2007.
19. B. Karlsson, Beyond the C++ Standard Library: An Introduction to Boost, Addison Wesley, Aug 31, 2005, p. 432.