Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

BUILDING DEVS MODELS WITH THE CADMIUM TOOL

Laouen Belloli

Computer Department
Universidad de Buenos Aires
Intendente Güiraldes 2160
Buenos Aires, C1428, ARGENTINA

Damian Vicino
Cristina Ruiz-Martin
Gabriel Wainer

Department of Systems and Computer Engineering
Carleton University
1125 Colonel by Dr.
Ottawa, ON, K1S 5B6, CANADA

ABSTRACT

Discrete Event System Specification (DEVS) is a mathematical formalism to model and simulate discrete-event dynamic systems. The advantages of DEVS include a rigorous formal definition of models and a well-defined mechanism for modular composition. In this tutorial, we introduce Cadmium, a new DEVS simulator. Cadmium is a C++17 header only DEVS simulator easy to include and to integrate into different projects. We discuss the tool's Application Programming Interface, the simulation algorithms used and its implementation. We present a case study as an example to explain how to implement DEVS models in Cadmium.

1 INTRODUCTION

Discrete Event System Specification (DEVS) is a mathematical formalism to model and simulate discrete-event dynamic systems (Zeigler et al. 2000). DEVS manages the complexity of the system using a modular structure. The system is decomposed using two types of models: (1) basic behavioral models called atomic models and (2) structural models called coupled models.

In (Vicino et al. 2015), we introduced a new environment derived from our experiences with the CD++ simulator (Wainer 2009), called CD-Boost, to develop and execute DEVS models. CD-Boost allows defining DEVS models using the C++11 standard (and the Boost library), and showed good speed results, outperforming all existing DEVS simulators. Nevertheless, an issue with this environment was the difficulty to define complex models, as it implements standard DEVS models without ports. When defining complex models, the user needs to define complex message data types that could include various attributes that might not be needed. This introduces extra overhead in terms of memory use, compile and execution time. Based on the experience with CD-Boost, we introduce an improved version, called Cadmium, which solves the above-mentioned issues and includes checking of the models before simulating them.

In this tutorial we will explain the key features of the simulator and how to use it to build DEVS models. Section 2 introduces the related work, including an explanation of DEVS and advances on DEVS simulators. In section 3, we present the Cadmium API and the new architecture. Section 4 introduces a case study to show how to define and execute DEVS models.

2 BACKGROUND

Discrete EVent System specification (DEVS) (Zeigler et al. 2000) is a hierarchical and modular formalism for modeling Discrete Events Systems (DES). The hierarchical and modular structure of DEVS allows defining multiple sub-models that are coupled together. The coupled model can also be used as a submodel, defining a multi-level hierarchical structure.

Atomic models define the behavior of the system. The formal definition of an atomic model is as follows:

$$AM = < X, Y, S, ta, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda >$$

Where:
X is the set of input events.
Y is the set of output events.
S is the set of sequential states.
$ta: S \rightarrow \mathbb{R}_0^+ \cup \infty$ is the time advance function that determines the time until the next internal transition.
$\delta_{ext}: QxX^b \rightarrow S$ is the external transition function that determines the next state when external events arrive, where $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ }, e is the elapsed time since the last state transition and X^b is a set of bags over elements in X.
$\delta_{int}: S \rightarrow S$ is the internal transition function that determines the state transition of the model when the state duration is over, and no external event has arrived.
$\delta_{con}: QxX^b \rightarrow S$ is the confluence transition function that determines the next state when and external events arrive at the same time than an internal transition is scheduled.
$\lambda: S \rightarrow Y^b \cup \emptyset$ is the output function that determines the output of the model based on its current state. Y^b is a set of bags over elements in Y and $\emptyset$ is the empty set.

Coupled models are defined connecting multiple DEVS models (coupled or atomic) linking the models' inputs and outputs. A coupled model is defined as the next 7-tuple:

$$CM = < X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC >$$

Where:
X: Is the set of input events.
Y: Is the set of output events.
D: Is the set of the names of the sub-components.
$\{M_d\}$: Is the set of sub-components where $d \in D$. Each M_d is a DEVS model (either atomic or coupled)
EIC: is the set of external input couplings
EOC: is the set of external output couplings
IC: is the set of internal couplings

The use of DEVS provides several advantages in the field of modeling and simulation. It is a methodology to develop hierarchical models in a modular way. This modularity allows model reuse and thus, reducing development time and testing. The model definition, implementation, and simulation are separated. The same model can be implemented on different platforms facilitating reliability and correctness.

Van Tendeloo and Vangheluwe (2017) provide an overview of the current state of eight different DEVS simulators, selected based on their functionality, DEVS compliance, and performance. *adevs* (a Discrete Event Simulator) is the fastest well stablished DEVS framework evaluated so far (Van Tendeloo and Vangheluwe 2017). In *adevs* (Nutaro 2014), coupled models are reduced to an equivalent atomic model 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 (Nutaro 2011) 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.

PyDEVS performance comes close to *adevs*. PyDEVS offers a modular architecture, using a *BaseSimulator* class to run both coupled and atomic DEVS models, as it applies symbolic flattening (Van Tendeloo and Vangheluwe 2014). It offers a variety of schedulers (sorted list, activity map, and a heapset) for speedup. PyDEVS supports dynamic typing; therefore, all the messages are not required to be of the same type. PyDEVS has sequential and distributed variants (Van Tendeloo and Vangheluwe 2014b) but both focus on computational activity information to reduce simulation time. New features (such as pause, resume, and step the simulation) have been added to help debugging (Van Mierlo et al. 2017).

Other DEVS simulators include VLE (Virtual Laboratory Environment) (Quesnel et al. 2007), which couples multiple simulators within a DEVS-Bus architecture and uses DEVS for coordination; CD++

(Wainer 2009), whose performance is slower than *adevs* but also implements Cell-DEVS and has a plugin for Eclipse; *MS4Me*, which allows building models using a custom natural-like language called DNL combined with Java; DEVS-Suite, the successor of DEVSJava.

Several of these simulators have been compared using the DEVStone synthetic benchmark (Glinsky and Wainer 2005). In (Wainer et al. 2011), *adevs* outperformed CD++ for large models. Flattening the model (Kim et al. 2000) modifies the simulator structure so that there is no hierarchical architecture, and the overhead induced by passing messages through various levels disappears. The impact of flattening has also been measured using DEVStone, showing clear performance improvements.

(Vicino et al. 2015) introduced a sequential architecture and an effective implementation of the abstract simulator. The performance was assessed by comparing it to *adevs* using DEVStone, and it showed that no new overhead was introduced. It outperforms *adevs* when there are numerous simultaneous events. Nevertheless, the models are complex to define, as the models do not use input/output ports, or different types of messages for different models, which results in complex model definitions, difficulties in verification of the models, and slow compilation (and massive use of memory).

3 CADMIUM

Based on our experience from (Vicino et al. 2015), we defined the Cadmium DEVS simulator, which solves the above-mentioned issues and includes checks of the models to help with verification. Cadmium is a C++17 header-only simulator easy to include and integrate into different projects. The simulator only uses iso-cpp standard code, which make it compatible with different platforms. It has been evaluated on Linux using GCC and Clang compilers, on Windows and FreeBSD. It is under the BSD open source license.

3.1 Application Programming Interface

At the user level, in Cadmium, we need to define a class for each type of atomic model and coupled model.

```
struct AtomicName_defs{ //Input/output Port declaration
  struct input_port1 : public in_port< MSGi1> {};
...
  struct output_portn : public out_port< MSGon> {};   };

template<typename TIME>

class AtomicName{
  using defs=AtomicName_defs;    //port definition in context

  public:
     struct state_type    { // your state variables here   };
       state_type state;
       AtomicName() noexcept { //parameters/initial state values here} ;

     void internal_transition() {
          //  internal transition function here  }
     void external_transition(TIME e, typename make_message_bags <input_ports>::type mbs) {
         //  external transition function here        }
     void confluence_transition(TIME e, typename make_message_bags <input_ports>::type mbs) {
        // confluence function here        }
     typename make_message_bags<output_ports>::type output() const {
       // output function here. Fill bags
    return bags;               }
    TIME time_advance() const { // time advance function here    }
 };
```

Figure 1: DEVS atomic model implementation using Cadmium.

The template presented in figure 1 provides the general structure of the class for defining the atomic models. It provides a constructor where the model parameters and the five DEVS functions can be defined.

As seen in the figure, first, we declare the model ports as a structure and the atomic model as a class. Each atomic model class has the set of state variables grouped together in a structure. It also has a model constructor to instantiate the model parameters and initial values. The user completes the template and implements all the DEVS functions in C++.

Once atomic models are defined, we use a class for defining coupled models, which uses model identifiers, the list of input ports, the list of output ports, the list of components, the lists of external input couplings, external output couplings and internal couplings, following the DEVS coupled model specification. They are implemented using the template provided in Figure 3 (which shows how to define the coupled model in Figure 2).

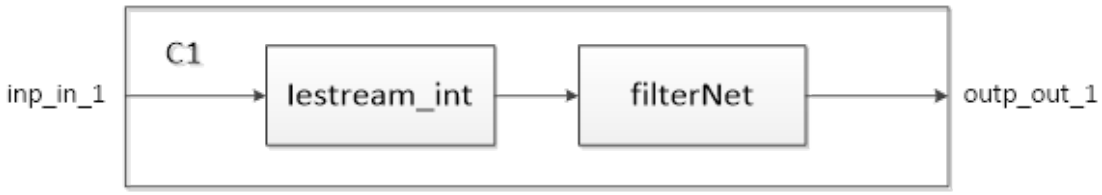

Figure 2: Example of a DEVS coupled model.

We first instantiate all the atomic models with a specific name. Then, we declare the coupled model ports. We then define the top model components: input ports, output ports, submodels, external input couplings, external output couplings and internal couplings. The coupled model is defined using all these components.

```
std::shared_ptr<cadmium::dynamic::modeling::model> filterNet_ins =
   cadmium::dynamic::translate::make_dynamic_atomic_model<filterNet,TIME> ("filterNet1");

 struct inp_in_1 : public in_port<int>{};         // input port
 struct outp_out_1 : public out_port<double>{};   // output port
 cadmium::dynamic::modeling::Ports iports_C1 = {typeid(inp_in_1)};  // ports types
 cadmium::dynamic::modeling::Ports oports_C1 = {typeid(outp_out_1)};

 cadmium::dynamic::modeling::Models submodels_C1= { iestream, filterNet_ins }; // top model

 cadmium::dynamic::modeling::EICs eics_C1= {    // External Input Couplings - EIC
   cadmium::dynamic::translate::make_EIC< inp_in_1, iestream_defs::in >("iestream1")  };

 cadmium::dynamic::modeling::EOCs eocs_C1 = {   // External Output Couplings - EOC
   cadmium::dynamic::translate::make_EOC<filterNet_defs::out, outp_out_1> ("filterNet1")  };

 cadmium::dynamic::modeling::ICs ics_C1 = {  // Internal Couplings - IC
    cadmium::dynamic::translate::make_IC< iestream_defs::out, filterNet _defs::in >  };

 std::shared_ptr<cadmium::dynamic::modeling::coupled<TIME>> C1 =
     std::make_shared<cadmium::dynamic::modeling::coupled<TIME>>("C1",  submodels_C1,
            iports_C1,  oports_C1,  eics_C1, eocs_C1,  ics_C1  );
```

Figure 3: DEVS coupled and top model implementation using Cadmium.

Once the model is defined, to run a simulation, call the simulator as shown in figure 4. The logger system is flexible. Here we exemplify how to log the simulation time and the messages in a file. For this case, the logger is defined declaring the name of the output file. The runner uses the time class and the logger as template parameters and the top model name as parameter. The simulator is call with the command "run_until" and simulation running time as parameter.

```
 static std::ofstream out_data("output_file_name.txt");
   struct oss_sink_provider {
       static std::ostream& sink()
       return out_data;
   };

using log_messages=cadmium::logger::logger<cadmium::logger::logger_messages,
      cadmium::dynamic::logger::formatter<TIME>, oss_sink_provider>;
using logger_top=cadmium::logger::multilogger<log_messages, global_time>;

cadmium::dynamic::engine::runner<NDTime, logger_top> r(TOP, {0});
r.run_until(NDTime("04:00:00:000"));
```

Figure 4: Logger definition and simulator call in Cadmium.

3.2 Simulation Algorithm and Implementation

We used the abstract simulator defined in (Vicino et al. 2015), with minor differences needed to add ports.

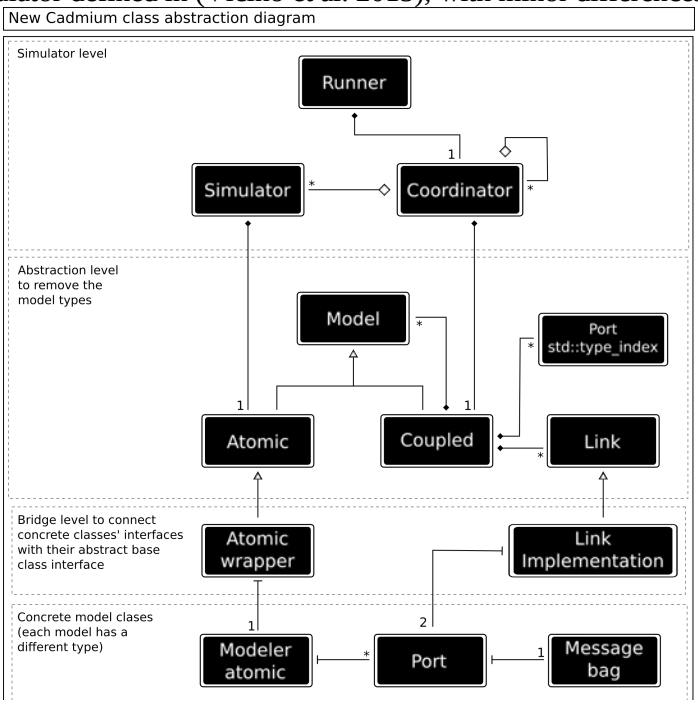

Figure 5: Cadmium architecture abstraction layers.

The proposed architecture, defined in Figure 5 has various levels:

- **Simulation:** The abstract simulator algorithm is implemented at this level.
- **Abstract modeling:** it defines the modeling class interfaces used by the simulation level classes.
- **Bridge:** it includes two model wrappers to connect the model and the abstract model classes. Then, each time a simulator class uses an abstract model, a bridge forwards calls a concrete model.
- **Concrete modeling:** In this level, all the concrete models are defined by the modeler.

The idea of two separated modeling levels (abstract and concrete) allows us to have strongly typed models in the concrete level, while using a single interface at the abstract level. Then, we can define different datatypes for the messages sent through the different ports and we can do static checks at compilation time. The bridge level maps the abstract modeling interface API with the real methods implementing that API in the concrete level. Following, we briefly discuss each of the levels.

3.2.1 Simulation Level

The simulation level is where all the modules in charge of running the simulation of the implemented model are defined. These modules implement the abstract simulators, a top-down implementation where the root coupled model sends advance requests 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 (using *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 routes the messages to advance the simulation once again, until the end.

The *Coordinator* class takes the *Coupled model* as a constructor parameter. *Coupled model* has a vector of submodels of type *Model* in the abstract level. These submodels are *Atomic* and *Coupled* model pointers derived from the class *Model*. For each submodel, the *Coordinator* tries to cast the *Model* into an *Atomic* model pointer. If the cast works, the *Model* is an *Atomic* model and a *Simulator* is instantiated to manage it. If the cast fails, the *Model* is a *Coupled* model and a *Coordinator* is recursively instantiated.

```
Coordinator
    Next, last, FEL; // Next/Last event, 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) )
        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 c to external_imminents
        add self.inbox contents to c.inbox
    if t == next then
        for each Coordinator c of a submodel receiving input from imminent i because IC
            set temp = collect_outputs(i)
            if not empty temp and c.next != t then
                add c to external_imminents
            add temp to c.inbox
    for each Coordinator c in Union(imminent, 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 6. The abstract simulator algorithm for coupled models.

In Cadmium a series of checks are executed before the simulation start. For coupled models, those checks are made in the Coordinator class constructor. The checks include: (1) Link types consistency: the model should not connect two ports with different message types; (2) Connected ports: the model should not connect invalid ports based on the EIC, EOC and IC structure; (3) Valid coupled and atomic submodels: we check recursively that the submodels are valid.

The **coordinator** class manages the abstract algorithm for coupled models by defining the next two functions: *advance_simulation* and *collect_outputs* (as shown in Figure 6) that serves to communicate with its parent coordinator or runner.

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.

The **simulator** class manages the abstract algorithm for atomic models by defining the next two functions: *advance_simulation* and *collect_outputs* as shown in Figure 7 that serves to communicate with its parent coordinator or runner.

```
Simulator
    next, last, model // // Next scheduled event time, last event, atomic model being simulated

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

Figure 7. The abstract simulator algorithm for atomic models

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.

Apart from the Simulator and Coordinator classes, a Runner 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

3.2.2 Abstract Modeling Level

We use abstract base classes without template parameters as interfaces, and we define the concrete template classes as derived from the abstract classes. Therefore, we can deal with the objects using their abstract base class without knowing their real type. The abstract modeling level defines the interface of all the atomic and coupled models the simulator uses. For this purpose, we define an atomic model interface, a coupled model interface, and a link model interface. To pool all the models together (atomic and coupled), we defined an abstract Model class and atomic and coupled classes are derived from the abstract Model class. Then, we can pass a pointer to a Model object that can either be an atomic or a coupled model.

The abstract atomic class defines virtual methods for the internal_transition, external_transition, output, confluence_transition, and time_advance functions. At this level, the output function returns a generic message bag where the real message types are hidden, and the obtained message has a generic type. This interface is used in the *advance_simulation* methods of the Simulator class.

The abstract coupled class takes a list of abstract models and three lists of abstract links for the EIC, EOC and IC connections. At this level, all the submodels are instances of the abstract Model class and the links are all instances of the abstract link class. For this purpose, the submodels and links are passed to the constructor using pointers (as required by C++ to use the base class instead of the derived class). Because *Coupled model* takes all the parameters as constructor parameters, no static checks can be made in compilation time, and they are performed at runtime. This is not a major problem because they are still done before the simulation starts, preventing crashes during the simulation.

The *Coupled model* class does not need to implement any method because, in DEVS, coupled models are only the model structure. Therefore, the *Coupled model* class only implements the interface requested by the *Model* class in the abstract layer. This is mandatory for both, *Coupled* and *Atomic models* in the abstract layer, so they can be passed as a parameter to other coupled models by using the *Model* interface.

The abstract link class is used to hide the real type of the link implementation class, which have the port type that as the message type. The link interface has a simulation API that allows the coordinator to request to the link to do the message routing between generic message bags to correctly add the collected messages in *collect_outputs* in the receiver models inboxes.

3.2.3 Bridge Level

We implemented an *Atomic Wrapper* class derived from the abstract *Atomic* class that takes the *Modeler atomic* model class type as a template parameter. The *Atomic Wrapper* implements the mapping between the abstract *Atomic* model class interface and the *Modeler atomic* model class interface. Because the *Atomic* Wrapper knows the *Modeler atomic* model type, it can implement all the following static checks:

- Model methods: we check whether all the methods of the atomic model (internal, external, time advance, output and confluence) are defined, and they have the correct parameters and return types.
- Ports: we check that the model port types are correct.
- Valid model state type: we check if the model class has an attribute called *state* of type *model* name::state type.

The *Atomic Wrapper* is derived from the *Modeler atomic* model class (i.e. the one defined by the modeler for each concrete atomic model) and from the *Atomic* class in the abstract level. Its main function is to map the concrete methods of the *Modeler atomic* model class into the methods of the *Atomic* class to connect both levels.

One major problem when implementing this abstraction is that the *Modeler atomic* functions (external_transition, confluence_transition, and output) have the message types in their definition as we allow different message types across the model. To solve this problem, the abstract interface implements the message bags using the *boost::any* type to allow messages of any type in the same container. *boost::any* allows us to convert any object into a *boost::any* object, the problem is that once the object is converted, we lose the real type, and to come back from the *boost::any* object to the real type we need to explicitly know the real type to cast the object. The A*tomic Wrapper* uses the real port types definition of the model to correctly cast the messages from the *boost::any* object to the correct message type.

To map *boost::any* messages into the correct port message type we use an *std:: map* with *std::typeindex* as the keys and *boost::any* as the values. *std::map* holds the messages that were converted into *boost::any* using their port type as the key. *std::type index* allows us to get a hashable representation of the port type that can be used as the map key. However, a major problem is that we cannot obtain the real type from them. Because of this, for each *std::type_index* key in the *std::map*, we must traverse the *std::tuple* with the model port types, get the *std::type_index* representation of the port type, and compare it with the key to find the correct port. Once we find the correct port, we use the port message type to cast the *boost::any* value into the correct message object. In this way, we can go back and forward from the *std::tuple* port structure to the abstract representation using *std::maps* of *boot::any*.

The resultant atomic model interface (i.e. *Atomic* in the abstract layer) is a class that defines the DEVS functions (external_transition, internal_transition, confluence_transition, output, and time_advance). This abstract class implements a mapping between the interface types and the concrete model types and then, it uses the concrete model methods to obtain the correct results. *Atomic* models in the abstract layer also implement the virtual *Model* class interface, so they can be treated as abstract models. The *Model* class interface only has getters for the model ids used by the *Simulator* and *Coordinators* to identify them.

3.2.4 Concrete Modeling Level

At the concrete modeling level, we have the *Modeler atomic* class defined by the modeler implementing the corresponding virtual methods of the *Atomic* model in the abstract level. The modeler must also define the concrete Port and Message bag classes as explained in section 3.1. These concrete classes are connected by the bridge classes to the abstract classes used by the simulator level classes.

3.3 Compilation Performance

To determine the compilation memory and time complexity of the new Cadmium architecture, we run different models that could not be compiled with the original architecture. We chose the models so that they allow us to separate the various aspects of a model to see their impact in the compilation memory complexity. We considered two scenarios:

- Scenario 1: add object instances, incrementing the number of atomic and coupled modes
- Scenario 2: add types to the model, incrementing the number of port types and atomic model types.

In the first test, we created a coupled including from 1 to 10000 identical empty atomic models. As we can see in Figure 8, we were able to compile models with over 10000 atomic models in less than 8 seconds and consuming no more than 800 MB of RAM. The time and memory complexity grow linearly with the number of atomic model objects inside the TOP model. Figure 8.a shows a stepped pattern in compilation time complexity, and that the general time complexity is linear.

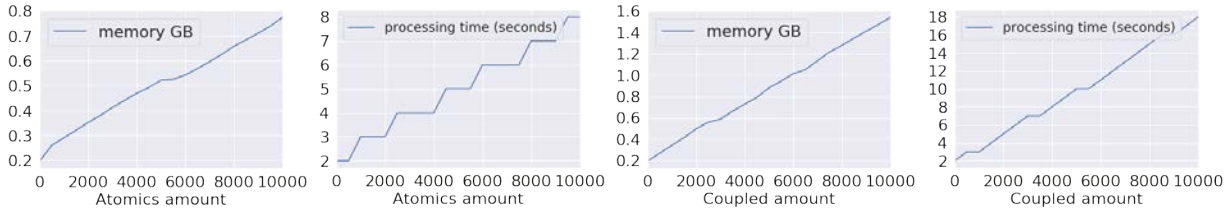

Figure 8: Compilation time (sec.) and RAM memory usage (GB) for the number of models' test

In a second test, we defined from 1 to 10000 empty coupled sub models. As we can see in Figure 8 (Coupled models), the number of coupled models also affects the compilation time and memory use linearly, but it is more expensive than compiling a model with multiple atomic models.

In the Scenario 2 we conducted two tests: we vary the number of port and atomic model types:

- **Port types**: We define an atomic model with different 1 to 120 output ports.
- **Atomic model types**: This test is like the number of atomic models' number test in Scenario 1, but in this case each atomic model has a different type.

As we can see in Figure 9, varying the number of port types and atomic model types have a linear impact in both the compilation time and memory complexity. This is a major improvement from the first version of Cadmium, where the atomic model types have impact in the compilation time and memory complexity. These results show a major improvement in the Cadmium model compilation time, from being unable to compile medium to large models to compile them in a few seconds.

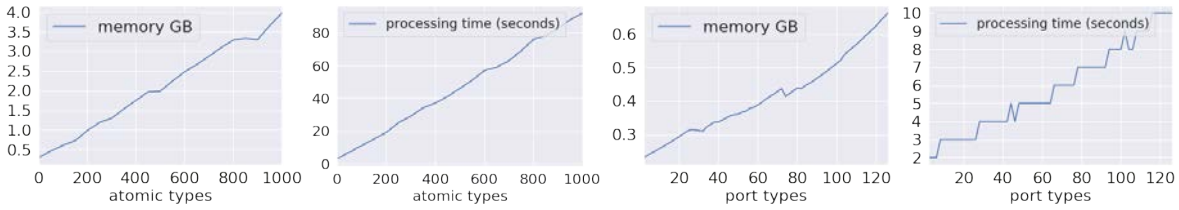

Figure 9: Compilation time (sec.) and RAM memory usage (GB) for the number of port/atomic types test.

Because the first version of Cadmium was based on model types, each model component needed to be declared explicitly without using any flow control system, as iteration cycles (for example a loop). This is also a major problem at compilation time, because it is much more efficient to parse a four-line loop that declares thousands of models, than parsing thousands of lines (one for each model). In the experiments, we did not use iteration flows to create any part of the model to assess just the performance improvements of new Cadmium architecture. However, we will be able to use them with this updated version.

4 EXAMPLE: ALTERNATE BIT PROTOCOL MODEL IN CADMIUM

We explain how to use Cadmium using an example of modeling and simulation of the Alternating Bit Protocol (ABP) (Wainer 2009). ABP is a network communication protocol that provides reliable transmission through an unreliable network. Each time the sender sends a packet, it waits for an acknowledgment.

If the acknowledgment does not arrive before a timeout, the sender resends the packet. To distinguish consecutive packets, the sender adds an additional bit on each packet (called alternating bit because the sender uses 0 and 1 alternatively).

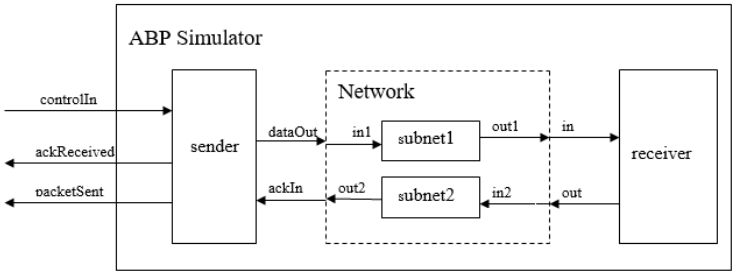

Figure 10: ABP Coupled model.

The ABP can be model using DEVS as a coupled model of 3 components (figure 10): sender, network and receiver. The network is decomposed further to two subnets corresponding to the sending and receiving channel, respectively. The behavior of the receiver is to receive the data and send back an acknowledgment extracted from the received data after a time. The subnets just pass the received data after a time delay. However, to simulate the unreliability of the network, only 95% of the data will be passed in each of the subnets, i.e. 5% of the data will be lost through the subnet.

The behavior of the sender is more complicated. Its state depends on the following user-defined state variables: *Alt_bit*, *sending*, *ack*, *packetNum*, and *phase:*

- The sender changes from initial phase *passive* to *active* when a *controlIn* signal is received.
- When it is in *sending* mod, it sends a packet plus an alternating bit once the *sending_time* is elapsed.
- The sender waits for the acknowledgment.
- If the *timeout* expires, the sender re-sends the previous packet with the alternating bit. If the expected acknowledgment is received before the *timeout*, the sender sends the next packet.
- The sender will go to *passive* phase when all packets have been sent out successfully.
- An output will be generated when a packet is sent out (*packeSent*, *dataOut*) or when an expected acknowledgment is received (*ackReceived*).
- For simplicity, the packet sent out by the sender is just the packet sequence number plus an alternating bit, (e.g. 11 for the first packet, 100 for the 10th packet, etc.).
- The *controlIn* signal is a positive integer indicating how many packets should be sent in a session.

To show how to use Cadmium, we will focus on the definition of one atomic model and the coupled model. The complete model implementation and the instruction on how to run the simulations are available in GitHub (https://github.com/SimulationEverywhere/ModelLibraryCadmium/)

The formal definition of the receiver atomic model is as follows:

Receiver = <S, X, Y, δint, δext, λ, ta>

S = {passive, active}
X = {in}
Y = {out}
δint (active) = passive
δext (in, passive) = active
δext (in, active) = active
λ(active)
    send "in" without the alternating bit to port "out" //extract the alternating bit and send back
ta(passive) = INFINITY
ta(active) = receiving_time

This formal definition is implemented in Cadmium in an hpp file as shown in figure 11. We first declare the ports with the type of message that they will manage. In this case, it is a C++ class called Message_t with a single attribute. Then, we define the atomic model: we define the model parameters, the constructor/s, the state, the input and output ports, and all the DEVS functions. For the constructor, we must define a default constructor that does not take any parameters, but we can add additional constructor if needed. The model state is defined as a struct called "state_type" with as many attributes as needed. In this case, we have two attributes: *ackNum* (representing the acknowledgment number) and *sending* (that represents if the model is in sending mode or not). For implementing the DEVS functions, we just need to translate the formal definition into C++ language using the Cadmium interface. Finally, we need to define how the model state will be logged.

```
struct Receiver_defs{ //PORT DECLARATION
    struct out : public out_port<Message_t> {};
    struct in : public in_port<Message_t> {};       };

template<typename TIME> class Receiver{ //ATOMIC MODEL DEFINITION
  using defs=Receiver_defs; //Setting ports defs

  public:
   TIME   preparationTime; //Model parameter

   Receiver() noexcept{ // default constructor
       preparationTime  = TIME("00:00:10");
       state.ackNum    = 0;
       state.sending     = false;           }

   struct state_type{ // state definition
      int ackNum;
      bool sending;              };

   state_type state;

   // ports definition
   using input_ports=std::tuple<typename defs::in>;
   using output_ports=std::tuple<typename defs::out>;
   void internal_transition() { // internal transition
      state.sending = false;         }

   // external transition
   void external_transition(TIME e,typename make_message_bags<input_ports>::type mbs){
    if(get_messages<typename defs::in>(mbs).size()>1) assert(false && "1 msg max");
       for(const auto &x : get_messages<typename defs::in>(mbs)){
          state.ackNum = x.value;
          state.sending = true;             }              }

   // confluence transition
   void confluence_transit(TIME e,typename make_message_bags<input_ports>::type mbs ){
       internal_transition();
       external_transition(TIME(), std::move(mbs));        }

   typename make_message_bags<output_ports>::type output() const { // output function
      typename make_message_bags<output_ports>::type bags;
      Message_t out;
      out.value = state.ackNum % 10;
      get_messages<typename defs::out>(bags).push_back(out);
      return bags;        }

   TIME time_advance() const {  // time_advance function
     TIME next_internal;
     if (state.sending) next_internal = preparationTime;
     else     next_internal = std::numeric_limits<TIME>::infinity();
     return next_internal;          }
};
```

Figure 11: Code snippet for Cadmium implementation of the receiver atomic model.

Once all the atomic models are defined, we need to instantiate them and implement the coupled model in a main.cpp file as we show in figure 12. We first specify the time type for the model. We can use a float or any other class that represents time. Second, we declare the ports for all the coupled models. Third, we specialize all the atomic models that are defined using a template system. In this case, we need to specialize the model that parses the input file with the message type. Finally, we define the main function that contains the logger definition, the atomic models' instantiation, the implementation of the coupled models and the call for the runner.

```
using TIME = NDTime; //Specify the time class
struct inp_control : public cadmium::in_port<Message_t>{}; // Set input/output ports
struct outp_ack : public cadmium::out_port<Message_t>{};
...

/* Specialize the atomic parser to send inputs to the top model*/
template<typename T> class ApplicationGen : public iestream_input<Message_t,T> {
   public:
    ApplicationGen() = default;
    ApplicationGen(const char* file_path) : iestream_input<Message_t,T>(file_path) {}     };

int main(int argc, char ** argv) {    /*************** Loggers ******************/
  static std::ofstream out_data("abp_output.txt"); //define the name output file name
    struct oss_sink_provider{ static std::ostream& sink(){  return out_data; }        };
  using log_messages=cadmium::logger::logger<cadmium::logger::logger_messages,
        cadmium::dynamic::logger::formatter<TIME>, oss_sink_provider>;

  /****** Instantiate atomic Parser ******************/
  string input_data_control = argv[1];
  const char * i_input_data_control = input_data_control.c_str();
  std::shared_ptr<cadmium::dynamic::modeling::model> generator_con =
    cadmium::dynamic::translate::make_dynamic_atomic_model<ApplicationGen, TIME, const
    char* >("generator_con" , std::move(i_input_data_control));

  ...     /****** Instantiate atomic Sender, Receiver and Subnets ******************/
  std::shared_ptr<cadmium::dynamic::modeling::model> subnet1 =
    cadmium::dynamic::translate::make_dynamic_atomic_model<Subnet, TIME>("subnet1");
  std::shared_ptr<cadmium::dynamic::modeling::model> subnet2 =
    cadmium::dynamic::translate::make_dynamic_atomic_model<Subnet, TIME>("subnet2");

  /******* Define Coupled Model Network ********/
  cadmium::dynamic::modeling::Ports iports_Network = {typeid(inp_1),typeid(inp_2)};
  cadmium::dynamic::modeling::Ports oports_Network = {typeid(outp_1),typeid(outp_2)};
  cadmium::dynamic::modeling::Models submodels_Network = {subnet1, subnet2};
  cadmium::dynamic::modeling::EICs eics_Network = {
    cadmium::dynamic::translate::make_EIC<inp_1, Subnet_defs::in>("subnet1"),
  ...
  /******* Define Top Model (i.e including the atomic parser model) ********/
  cadmium::dynamic::modeling::Ports iports_TOP = {};
  cadmium::dynamic::modeling::Ports oports_TOP = {typeid(outp_pack),typeid(outp_ack)};
  cadmium::dynamic::modeling::Models submodels_TOP = {generator_con, ABPSimulator};
  cadmium::dynamic::modeling::EICs eics_TOP = {};
  cadmium::dynamic::modeling::EOCs eocs_TOP = {
  cadmium::dynamic::translate::make_EOC<outp_pack,outp_pack>("ABPSimulator"),
    cadmium::dynamic::translate::make_EOC<outp_ack,outp_ack>("ABPSimulator")      };
  cadmium::dynamic::modeling::ICs ics_TOP = {
    cadmium::dynamic::translate::make_IC<iestream_input_defs<Message_t>::out,
       inp_control> ("generator_con","ABPSimulator")      };
  std::shared_ptr<cadmium::dynamic::modeling::coupled<TIME>> TOP =
    std::make_shared<cadmium::dynamic::modeling::coupled<TIME>>(
      "TOP", submodels_TOP, iports_TOP, oports_TOP, eics_TOP, eocs_TOP, ics_TOP     );

   cadmium::dynamic::engine::runner<NDTime, logger_top> r(TOP, {0});
    r.run_until(NDTime("04:00:00:000"));                                          }
```

Figure 12: Code snippet for atomic model instantiation and ABP coupled model implementation.

We show the standard logger definition, but customized loggers can be defined. The atomic models are instantiated with the command "cadmium::dynamic::translate::make_dynamic_atomic_model", the atomic model identifier (i.e. a unique name) and the parameters for the constructor. For each coupled model, we need to declare, as arrays, the sub-models, input ports, output ports, external input couplings, external output coupling, and internal coupling. The coupled models are defined with the command "std::make_shared<cadmium::dynamic::modeling::coupled<TIME>>" as an array of the six previous elements preceded by the coupled model identifier. After implementing all the coupled model, we define the runner as "cadmium::dynamic::engine::runner<NDTime, logger_top> r(TOP, {0})", where NDTime represents the time type, logger_top is the logger we have selected, TOP is the name of the variable where the Top Model is defined and {0} is the simulation starting time. Finally, we call the runner with the simulation finishing time as a parameter.

The model implementation along with the simulator is compiled using gcc or clang to generate the executable. Different simulations can be run using different input files. In figure 13, we show the simulation log file for a scenario where the sender sends a single message that has 20 packets starting at time 10s. As we can see in the log file, packet 8 generated at time 00:03:22:000 was lost (i.e. no acknowledgment was received), therefore it was resent after the acknowledgment window was elapsed.

```
00:00:10:000 [iestream_input_defs::out: {20}] generated by model generator_con
00:00:20:000 [Sender_defs::packetSentOut: {1}] generated by model sender1
00:00:36:000 [Sender_defs::ackReceivedOut: {1}] generated by model sender1
00:00:46:000 [Sender_defs::packetSentOut: {2} generated by model sender1
00:01:02:000 [Sender_defs::ackReceivedOut: {0}] generated by model sender1
00:01:12:000 [Sender_defs::packetSentOut: {3}] generated by model sender1
00:01:28:000 [Sender_defs::ackReceivedOut: {1}] generated by model sender1
00:01:38:000 [Sender_defs::packetSentOut: {4}] generated by model sender1
00:01:54:000 [Sender_defs::ackReceivedOut: {0}] generated by model sender1
...
00:03:22:000 [Sender_defs::packetSentOut: {8}] generated by model sender1
00:03:52:000 [Sender_defs::packetSentOut: {8}] generated by model sender1
00:04:08:000 [Sender_defs::ackReceivedOut: {0}] generated by model sender1
00:04:18:000 [Sender_defs::packetSentOut: {9}] generated by model sender1
...
00:10:04:000 [Sender_defs::packetSentOut: {20}] generated by model sender1
00:10:20:000 [Sender_defs::ackReceivedOut: {0}] generated by model sender1
```

Figure 13: Simulation log snipped.

The simulation log files can be processes using different techniques to identify, for example, what is the percentage of packets lost, what is the average attempts needed to send a packet, etc.

5 CONCLUSIONS

We presented Cadmium, a DEVS simulator that builds over the simulator presented in (Vicino et al. 2015). Cadmium solves the two main disadvantages of that simulator: (1) lack of ports and (2) the restriction of using a single type of data for all the messages. Cadmium also implements other advanced features that help with model verification and reduces the probability of running a simulation with a bug. It implements model checks for both atomic and coupled models. Some of them are in compilation time and others in runtime initialization, it means that all of them are performed before a simulation starts. For atomic models, it checks that all the methods of the atomic model are defined, that the model port types are correct, and that the model has an attribute called *model name::state type* where the state of the atomic is defined. For coupled models, Cadmium checks that there are no connections between ports with different message types, that there are not invalid connections based on the model EIC, EOC or IC link structure, and that all the submodels inside a couple are valid (i.e. it performs atomic model checks).

The modular architecture of the simulator in (Vicino et al. 2015) is maintained as well as the simulation algorithms and its sequential execution. Therefore, a similar performance is expected. We keep the modular

architecture because it preserves the natural structure of DEVS models.

The new simulator was implemented as a library written in C++, compliant with the C++17 and the Boost library coding standard. It supports multiple data types for the Time, and Messages, and compiles in multiple platforms, including Linux, Windows, FreeBSD, and OS X. Cadmium is available on GitHub (https://github.com/SimulationEverywhere/cadmium).

REFERENCES

Belloli, L., G. Wainer, and R. Najmanovich 2016. "Parsing and Model Generation for Biological Processes." in *Proceedings of the Symposium on Theory of Modeling and Simulation (TMS-DEVS*). IEEE. (art. 21) Pasadena, CA, USA.

Chow, A.C., B. P. Zeigler and D. H. Kim 1994, "Abstract Simulator for the Parallel DEVS Formalism". *Proc. of the Fifth Conference on AI, Simulation, and Planning in High Autonomy Systems*, (pp. 157-163) Gainesville, FL

Kim, K., W. Kang, B. Sagong and H. Seo, 2000 "Efficient Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-Hierarchical One," in *Proceedings of the 33rd Annual Simulation Symposium*, (pp. 227-233) Washington, DC.

Nutaro, J. 2011 *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Hoboken, NJ: Wiley.

Nutaro, J. 2014. *A Discrete Event System Simulator*. [Online]. Available: http://web.ornl.gov/~1qn/adevs/adevs-docs/manual.pdf. Accessed: 22/04/2019

Quesnel,G., R. Duboz, E. Ramat and M. K. Traore, 2007 "VLE: a Multimodeling and Simulation Environment," in *Proceedings of the 2007 Summer Computer Simulation Conference*, (pp. 367-374) San Diego, CA

Ruiz-Martin, C., G. Wainer, and A. Lopez-Paredes 2018. “Formal Abstract Modeling of Dynamic Multiplex Networks”. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (pp. 61-72). ACM. Rome, Italy.

Van Mierlo, S., Y. Van Tendeloo, and H. Vangheluwe. 2017 "Debugging Parallel DEVS." *Simulation* vol. 93 no.4 pp: 285-306.

Van Tendeloo, Y., and H. Vangheluwe, 2014 "The Modular Architecture of the Python (P)DEVS Simulation Kernel” In *Proceedings of the Symposium on Theory of Modeling & Simulation*, (art. 14) Tampa, FL, USA

Van Tendeloo, Y., and H. Vangheluwe,2014b "Activity in PythonDEVS," *ITM Web of Conferences*, vol. 3, p. 01002.

Van Tendeloo, Y., and H. Vangheluwe, 2017. "An Evaluation of DEVS Simulation Tools." *Simulation* vol.93 no.2 pp 103-121.

Vicino, D., D. Niyonkuru, and G. Wainer 2015 “Sequential DEVS Architecture” *Proceedings of the Symposium on Theory of Modeling & Simulation*, (pp. 165-172) Alexandria, VA, USA

Wainer, G. 2009. *Discrete-event modeling and simulation: a practitioner's approach*. CRC Press, Boca Raton, FL, USA

Wainer,G., E. Glinsky and M. Gutierrez-Alcaraz, 2011 "Studying Performance of DEVS Modeling and Simulation Environments using the DEVStone Benchmark," *Simulation*, vol. 87, no. 7, pp. 555-580.

Zeigler, B.P., H. Praehofer and T. G. Kim 2000 *Theory of modeling and simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, San Diego, CA: Academic Press

AUTHOR BIOGRAPHIES

LAOUEN BELLOLI is a Ph.D. student at the Laboratory of applied artificial intelligence (LIAA) at the Department of Computer Science at University of Buenos Aires. His email is laouen.belloli@gmail.com.

DAMIAN VICINO has obtained a co-joint Ph.D. in Computer Science (Université de Nice-Sophia Antipolis) and Systems and Computer Engineering (Carleton University). Currently, he is a Software Developer Engineer at Amazon working in Alexa product. The research contribution of Dr. Vicino was done prior to joining Amazon His email address is damianvicino@cmail.carleton.ca

CRISTINA RUIZ-MARTIN has obtained a Ph.D. in Industrial Engineering (University of Valladolid, UVa) and Systems and Computer Engineering (Carleton University). She is a Postdoctoral Fellow at the Department of Systems and Computer Engineering at Carleton University. Her email address is cristinaruizmartin@sce.carleton.ca.

GABRIEL WAINER is a Full Professor at the Department of Systems and Computer Engineering at Carleton University. He is a Fellow of SCS. His email address is gwainer@sce.carleton.ca.