

Technical report for a DEVS formalism simulator in C++

Bc. Jozef Méry

xmeryj00@vut.cz

ABSTRACT

This technical report is concerned with a *DEVS formalism simulator* written in C++ as a part of the project assignment for the Simulation Tools and Techniques (SNT) subject in 2023. The report presents the formal definitions of *atomic* and *compound DEVS* models. Next, the report shares some details about the *library implementation*, and finally, it presents a *demo application* and discusses its *examples*.

1 Introduction

The project aimed to create a *DEVS formalism simulator* in C++ using the *standard library* only and provide an example from the *queue theory* area. This technical report introduces the *DEVS formalism* in this section, describes the simulator implementation in section 2, discusses the demo application and its examples in section 3, and finally, the report is concluded in section 4.

The *DEVS* (Discrete Event System Specification) formalism is a structure defined by Ziegler et al. [2] for modeling and analyzing systems with *discrete events*. There are two model types in *DEVS*, namely *atomic* and *compound* models. The atomic model is the smallest *standalone unit* in *DEVS* and is defined as follows:

$$M = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta), \quad (1)$$

where X is the set of *input* values, Y is the set of *output* values, and S is the set of *states*. These sets are often *structured* as $X = \{(x_1, x_2, \dots, x_m) | x_1 \in X_1, x_2 \in X_2, \dots, x_m \in X_m\}$. Individual members in the structured input and output sets are referred to as *ports* (the provided example has m ports). Next, δ_{int} is the *internal transition* function ($\delta_{int} : S \rightarrow S$), δ_{ext} is the *external transition* function ($\delta_{ext} : Q \times X \rightarrow S$, where Q is the *total state* defined as $Q = \{(s, e) : s \in S, 0 \leq e \leq ta(s)\}$, where e is the *elapsed time* since the last state transition), λ is the *output function* ($\lambda : S \rightarrow Y$), and finally, ta is the *time advance* function ($ta : S \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$), defining the *remaining* time until the next internal transition. [1][2]

The *compound DEVS* model is defined as follows:

$$N = (X, Y, D, \{M_d | d \in D\}, \{I_d | d \in D \cup \{N\}\}, \{Z_{i,d} | d \in D \cup \{N\}, i \in I_d\}, Select), \quad (2)$$

where X is again the set of *input* values, Y is the set of *output* values, D is the set of *atomic DEVS component references* (names), $M_d | d \in D$ is the set of *atomic DEVS components* (see definition 1), $I_d | d \in D \cup \{N\}$ is the set of *influencers* without a self-loop ($d : I_d \subseteq D \cup \{N\}, d \notin I_d$), $Z_{i,d} | i, d \in D \cup \{N\}$ is the set of *influencer transform functions* defining *connections* between components. Components can be *connected* to any other components or the enclosing compound model N according to the following *rules*:

$$Z_{N,d} : X \rightarrow X_d | d \in D, \quad Z_{i,N} : Y_i \rightarrow Y | i \in I_N, \quad Z_{i,d} : Y_i \rightarrow X_d | d \in D, i \in I_d, \quad (3)$$

implying that the components cannot directly *send inputs* to or *receive outputs* from the compound model N . Instead, they contribute to the *compound input* X and *compound output* Y . Finally, *Select* is a *tie-breaking function* ($2^D \rightarrow D$) in case of *concurrent events*. Additionally, it is *formally proven* that every *compound DEVS model* N is an *atomic DEVS model* simultaneously. Thus components can be *arbitrarily nested*, allowing the modeling of *hierarchical systems*. [1][2]

2 Library description

The library is written in *C++17* and is provided as a *single-file, header-only* library for easier potential installation. It implements the fundamental elements of *DEVS*, namely *atomic models*, *compound models*, and a *simulator* based on the *next-event* (calendar) algorithm in an *object-oriented* fashion. Additional features include *printing*, *simplified random number generation*, *helper constants*, or *helper types*. All library code is located in the *Devs* namespace.

The core element of the simulator is a *calendar structure* using the *next-event algorithm*. The implementation utilizes the *priority queue* from the standard library specialized with a custom *Event structure*. The calendar has only two priority hierarchy levels: *remaining time* and *FIFO*. Further priority hierarchy can be achieved using the *Select function* of compound models (see definition 2). There is always only *one calendar* in a simulator, without any additional *coordination mechanisms*.

Although the library *API* attempts to mimic the formal structures as closely as possible, there are some fundamental differences. For example, the model structures *lack* explicit input and output set parameters at run-time. However, they are present implicitly as the parameter of δ_{ext} and the output of λ , respectively (see definition 1). Furthermore, the input and output parameters are also expressed as *template parameters*. In fact, almost the entire library relies on *template meta-programming*, allowing the inputs, outputs, and states to be of almost any *built-in* or *user-defined type*. While there are no input and output type requirements, the state type needs to be *copy-able* (due to the definition of delta functions) and *printable* using the overloaded *bit-shift operator*: \ll . Another missing run-time parameter is the set of influencers IDs from definition 2. Instead, the influencers are *deduced* from the set of *influencer transform functions*. Furthermore, when defining the influencer transforms, the model name or transformer function can be *omitted* (by default constructing an optional: `,` or using `std::nullopt`). If the model name is omitted, the encapsulating compound model *N* is referenced (this way, the components do not need to know the compound model name). If the transformer is omitted, an *identity function* is used (i.e., the value is unchanged). Another difference is that the *Select function* (see definition 2) receives a *vector* of model names instead of a set, which trivializes the *FIFO selector* implementation. This modified *Select* may be defined as $\text{Select}': (2^D, <) \rightarrow D$, where $<$ is an *ordering relation*.

One of the significant *challenges* of the library implementation is the *arbitrary connection* and *nesting* of models due to the nature of *C++* itself. Both problems are solved by applying *indirection*. The arbitrary model connections could be solved by passing around a *generic pointer* cast to the desired type by the consumer. The *issue* with the *raw pointer reinterpretation* approach is that the behavior is *undefined* if the *typecast* is invalid. Instead, the custom *Dynamic class* is used, which is essentially a *thin wrapper* around *smart pointers* from the standard library. The core idea is to use *C++'s dynamic cast* instead of a *raw pointer reinterpretation* due to its ability to *detect* and *report* invalid casts. Using *dynamic casting* keeps the behavior *defined* even if the library user makes a *mistake* in the model connections, and the user can be notified appropriately. Thanks to *implicit conversions* and *operator overloading*, using the *Dynamic class* is almost *syntactically free* (although, in some cases, explicit typing may be required due to deduction failing/being incorrect). The issue of *arbitrary model nesting* is solved by defining a model *interface* named *IOModel* that defines common operations, e.g., *event handling*, *I/O*, etc., and then providing *distinct implementations* for both *atomic* and *compound models*.

As the project is limited to using only the standard library, only a *CLI* application is *feasible*. Due to this, an important additional feature is provided in the library, namely *printing*, located in the *Printer* namespace. Printers can *hook* into various events and inform the user accordingly. These events include *simulation start*, *simulation step*, *simulation end*, *time advanced*, *event scheduled*, *event action about to be executed*, and finally, *model state changed*. Three printers are provided in the library, namely a *base printer* (a mute printer that also serves as the *printer interface*, useful for models with an unreasonable amount of output), a *verbose printer* (without any text decoration, helpful when printing into a file), and finally a *decorated verbose printer* (suitable for printing into the *CLI* directly). The text decorations are based on *ANSI escape sequences*, and the library provides simple abstractions around them.

Additional minor features include *opinionated* abstractions for *random number generation* with *uniform*, *exponential*, and *Poisson distributions*. Next, *helper constants* and *types* are defined, e.g., *infinity* or *null type* (empty class without any special meaning within the library).

The library *API* aims to be as simple and minimal as possible, which means that many structures support *brace-initialization* and utilize *implicit conversions* or default values wherever appropriate. Unfortunately, parameter deduction does not work correctly in every situation, and explicit parameters may be required.

3 Demo application and provided examples

The project contains an *executable demo application* with *six examples* built using the created library. Two examples are *minimal* or *empty models* (one *atomic*, one *compound*) that showcase the minimal code required for running a simulation (a sort of "Hello world"). Furthermore, these models may be a *foundation* for creating *complex* examples. Next, there is a *traffic light* example simulation, the primary purpose of which was *testing* and *developing* the library. It is an *atomic model* with a *structured state*, *six input messages* (power and mode setting/toggling), and *one output message* (possibly the color the traffic

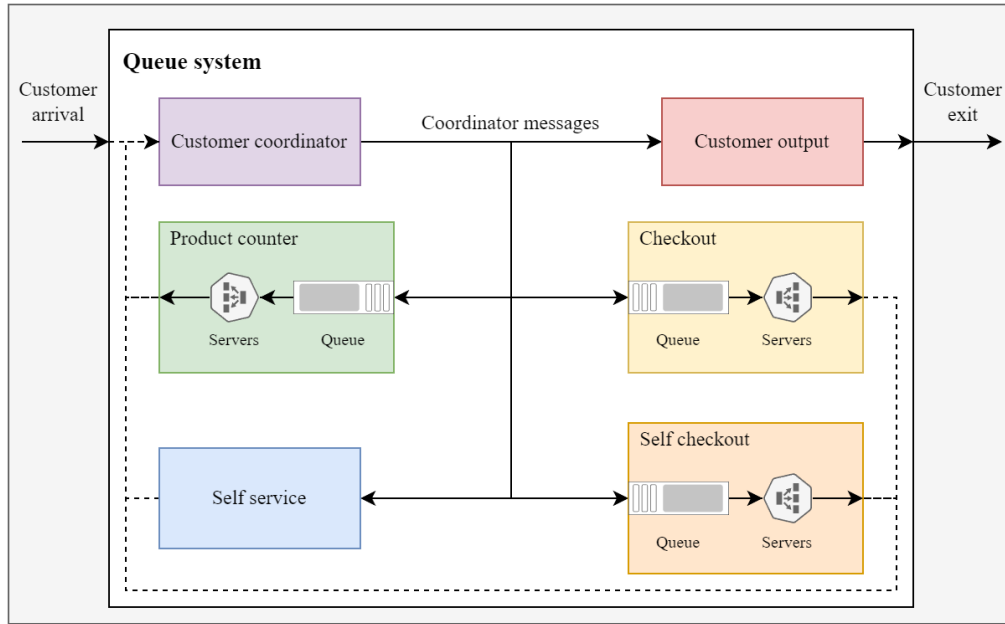


Figure 1. A block diagram representing a queue system from the provided examples (see section 3).

light is about to switch to).

The final three examples are from the area of *queue theory*. All three examples use the *same queue system model* (depicted in figure 1) with *similar parameters*. Before further description, it should be noted that the later referenced *product counter* is *not* a system counting products but a *service station* where customers may request certain additional products from servers (note that this is abstract and product types are not modeled). The queue system is a *compound grocery store model* with *four different customer types* (requests), *four different service stations*, and *one type of output message*, built using six atomic components.

The incoming customers of the queue system have *two independent requirements* (yielding four unique combinations), namely *product counter* (customers may wish to be served at the product counter station) and *age verification* (customers may need to have their age verified at checkout for certain products, e.g., *alcoholic beverages*, possibly *increasing* the service time). The customer arrivals are scheduled *before* the simulation with *exponentially distributed* durations between arrivals. The *flow* of customers is handled by the *customer coordinator* component that receives every new customer and every customer served by one of the stations. First, the customer is sent to the *product counter* if requested. Next, every customer *unconditionally* enters a special *self-service station*, representing customers picking products from shelves. The self-service station is an additional station *without* a queue and with *virtually infinite servers* (the customer is its server, and service starts immediately). The purpose of the *self-service* station is to introduce more complexity into the model. After self-service, customers proceed to one of the *checkout* stations based on the current *state* of their respective *queues*. Before a customer enters one of the checkouts, the coordinator component sends a *query message* to the checkout stations. Upon receiving responses from both, the customer is sent to the station with a *smaller queue* or *always the checkout* if the queues are *equally sized*. Finally, after checkout, the customer is *removed* from the queue system by the *customer output* component. The purpose of the customer output component is *filtering*, *unwrapping*, and *outputting* the customer from the coordinator message.

While the *product counter* station is a simple component without additional behavior, *checkout* and *self-checkout* stations are more *complex*. Specifically, checkouts handle the *age verification* request of customers (each slightly differently) and may have *errors*. The advantages of *classic checkouts* are a considerably *higher service rate*, *lower error probability*, and that age verification *does not increase* service time (the worker is already there; thus, the verification is part of the *regular service rate*). The possible disadvantages of classic checkouts are *costs* (building, maintaining), requiring a *human worker* for *each server*, *footprint*, *lower error handling rate*, etc. In contrast to checkouts, *self-checkouts* are more *space-efficient*, *cost-efficient*, *human-workforce-efficient*, and have *faster error handling* (they are better prepared for error handling as more errors are expected due to increased customer interaction). On the other hand, the downsides of self-checkouts include significantly *increased error probability*, *lower service rate*, and *age verification delays service* (the human worker may only sometimes be available).

The *purpose* of the queue system model and simulation could be finding the *ideal number of servers* for every station, or

investigating the ratio of checkout, and self-checkout servers, due to their differing properties and requirements. However, the demo application contains only *fictional parameters*; therefore, the experiments are tailored towards *differences between variants, predictability of behavior, or simulation run-time performance*. As mentioned previously, there are three different but similar parameterizations of this model in the demo application, specifically:

1. Short duration - In this case, the simulation lasts *ten minutes* and uses the colored verbose printer (additionally, exiting customers print the time of exit); thus, writing the output into a text file is unsuitable. During this time, around 15-20 customers are expected to arrive. This model is meant to showcase the *step-by-step execution* of models and present simulation issues caused by having too few customers.
2. Long duration - The parameters of this simulation are *identical* to the *short variant*, except instead of 10 minutes, the simulation lasts *ten days*, i.e., a *scaling factor* of 1440.
3. Large scale - The simulation window for this variant is *twenty-four hours*. Furthermore, the arrival rate of customers and the number of servers at every station from the short variant parameters is multiplied by a factor of ten, again yielding a total *scaling factor* of 1440.

Running any queue variant prints *statistics* for every station (except self-service). Simulating the *short variant* yields vastly *differing results*. The *execution time*¹ is anywhere between 500 milliseconds and 5 seconds. The *issue* with this simulation is the *time window*. Specifically, the time window needs to be *longer* in relation to the customer arrival rate. Only about 15-20 customers arrive in this simulation, which is *inadequate* for overcoming the randomness of the simulation, contributing to the *instability of results*. Furthermore, the time window also needs to be *longer* in relation to station service rates. Often, customers remain *unserved* in one of the stations (although the total busy times do reflect pending customers, meaning that busy times may sometimes be over 100%, indicating that the server had to work *overtime*). Another negative factor is the *amount of text output*, further slowing the execution. Next, the *long variant* has the *same parameters* as the short, except for the *ten-day time window*. The *execution time* of this simulation is consistently close to 3.5 seconds. During this time, around 24 000 customers are served in the system, almost precisely 1440 times more than in the short variant, indicating *correct scaling*. Due to a large number of served customers, the randomness becomes *insignificant*, yielding more stable results. Finally, the *large variant* yields *comparable* results to the long variant, with one *downside*. Even though the total scaling factor for both is 1440 in relation to the short variant, the *execution time* for the large variant is consistently over 8 seconds, i.e., more than double compared to the long variant, likely due to the *higher number of servers*, noticeably slowing down state updates.

In conclusion, the results of the *short variant* are *unpredictable* due to a *short time window*. The only significant *difference* between the *long* and *large* variants is the *execution time*, with the large variant requiring more than *twice* the time of the long variant, making the *long variant* the *best choice*, given the *fictional parameters* and *implementation details*. Both long and large variants yield relatively *stable* and *predictable results*. For example, the *product counter* has half the service rate of the total arrival rate, compensated by two servers, implying that, on average, the product counter should be able to handle *all incoming customers*. However, only about 75% of customers request service at the product counter. Thus, the product counter should be *busy* for around 75% of the total duration and *idle* for the remaining 25%. Furthermore, *checkouts* are *slightly busier* than *self-checkouts*, and the average queue size of checkouts is larger by about 0.5, caused by always selecting the checkouts when the queue sizes are *equal*. Simulations of the long and large variants consistently yield the *expected results* (max. 1% error).

4 Conclusions

Firstly, this technical report introduces the *DEVS formalism* and its *atomic* and *compound models* in section 1. Next, in section 2, features and essential *implementation details* of the library are discussed. Although the library attempts to *mimic* the formal definition as closely as possible, there are *minor differences*. Another aim of the library is to be *general*, covering most *use cases* while remaining *unopinionated*. *Generality* is achieved using *template meta-programming*, allowing any input, output, or almost any state type (state needs to be printable using the « operator and *copy-able* due to the definition of δ functions of atomic models). This generality also means that the library *does not provide* abstractions for these types, making the examples rather *lengthy*. However, designing a library for these types is *beyond this project's scope*. Finally, the provided demo application and its examples are presented in section 3. Three of the six provided examples are from the *queue theory area*, sharing the *same model* with slightly different parameters. Variants of the parameterization are referred to as *short* (short time windows), *long* (very long time window), and *large* (medium time window, scaled arrival rates, scaled number of servers, implying more customers present in the system on average). *Experiments* indicate that the best variant is *long*, possibly caused by *implementation details*. The project hopefully provides a good *foundation* and *potential* for additional work, e.g., *saving/loading of models* into/from files, *improved visualization*, or *acceleration using threads*.

¹The provided execution times are from running on a laptop with an i9-12900H CPU, Windows 11 x64, in a WSL environment running Ubuntu 20.04.4, measured using the standard library *chrono* module.

References

- [1] Petr Peringer. “SNT — Simulační nástroje a techniky”. Brno, 2023.
- [2] Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. *Theory of modeling and simulation: discrete event and iterative system computational foundations*. 3rd ed. London: Academic Press, 2019. ISBN: 978-0-12-813370-5.