DVCompute++ Simulator Tutorial

David E. Sorokin <david.sorokin@gmail.com>,
Yoshkar-Ola, Mari El, Russia

June 14, 2021

Introduction

The discrete event simulation model describes an activity that can be represented in terms of computations [1]. We define some simple units to represent primitive computations. Then we combine these units to create computations that would depend on results of the intermediate computations. This is a very general concept that came from functional programming world, but it can be applied for discrete event simulation too.

Moreover, in the same manner we define both sequential and distributed simulation models. While you can launch the former models on your laptop, the optimistic Time Warp method and the alternative conservative one will allow you to create large distributed simulation models that can be launched on super-computers. Everywhere the same approach works, the same idea is applied.

As the author of this method and the corresponding implementation DVCompute++ Simulator, I hope you will find this approach not being only as a theoretical invention, but also you will be able to create real and useful models for practice. Here I suppose that the reader is familiar with the C++ programming language.

1 Time Delay

The key point is how we represent time delays in simulation. So, we can hold the discontinuous process for the specified time interval and then proceed with the computation.

```
Process<Unit, auto> hold_process(double dt);
```

Here the function returns some instance of the `Process` data type, which is described right after this paragraph. Had the instance been involved in the simulation, the corresponding process would be delayed, while other simulation activities would continue their execution concurrently. But to involve this object in the simulation, we have to create and run a combined computation that would depend on it.

2 Process Data Type

The `Process` data type is a template of two type parameters:

```
template<typename Item, typename Impl = internal::process::BoxedImpl<Item>>
class Process;
```

The first type parameter `Item` denotes the data type, which value is returned by the `Process` computation. For example, `Process<int, Impl>` means that the discontinuous process computes the integer value.

Usually, you should not care about the second type parameter `Impl`. In most cases this type is automatically inferred by the C++ compiler. Therefore, I used the `auto` keyword in the pseudo-code, when stating the `hold_process` function above.

In general, the rule is as follows. If the `Impl` type parameter denotes something different from the default `internal::process::BoxedImpl` data type, then it literally means that the object, representing the discontinuous process, is probably allocated on the stack of the computer. This is very important for the performance! But if you see that the `Process` data type is parameterised by the default `Impl` data type, then it means that the `Process` instance data are allocated in the dynamic memory.

Besides, any `Process` instance can be transformed to another `Process` instance, where the second type parameter becomes default. In other words, we can reallocate the instance data from stack to dynamic memory. For example, this is important for creating recursive computations.

```
template<typename Item, typename Impl>
class Process {
  ...
  operator Process<Item>() &&;
};
```

I will call such an instance of the `Process<Item>` data type with the default `Impl` data type a *boxed representation* of the object, which means that the `Impl` data type is erased and the corresponding object data are allocated in the heap.

Two ampersands `&&` mean that the source object is *moved*, i.e. consumed, which means that the source object cannot be used anymore. In the code it can look something like this:

```
Process<Item, Impl> comp = ...;
Process<Item> boxed_comp { std::move(comp).operator Process<Item>() };
```

3 Side Effects

DVCompute++ Simulator is written in the functional programming style. Side effects are defined as computations, which are actually templates, where we have to define some data type to denote what values such a computation may return. In most cases, the side effect should return the fact itself that the effect is performed. It would be ideal to use the `void` data type instance, but C++ prohibits this.

Therefore, a dummy data type is defined, which single purpose is to parameterise the `Item` data type of some computation by such a type, which has only a single value that we can ever create.

```
struct Unit {
  constexpr Unit() = default;
};
```

The delay function `hold_process` returns an object of the `Process<Unit>` data type, or more precisely, the `Process<Unit, Impl>` data type for some `Impl`, which is inferred by the C++ compiler from the definition of this function. For brevity, I already used the pseudo-code data type `Process<Unit, auto>`.

4 Creating Processes

If you are familiar with the concept of `std::experimental::future` in C++, then you will find this material familiar too, especially if you applied the `then` method. Such methods and functions are also called *combinators*.

To create a primitive `Process` computation by the specified pure value, you can call one of the following functions (the pseudo-code):

```
template<typename Item>
Process<Item, auto> pure_process(const Item& item);

template<typename Item>
Process<Item, auto> pure_process(Item&& item);
```

For example, the `pure_process(int { 10 })` computation does nothing but *computes* the value of 10 when used in the simulation. But it is difficult to understand why we actually need this unless we introduce the following combinator, which has a long history in the world of programming.

To combine the current computation with its continuation, we can use the closure:

```
template<typename Item, typename Impl>
class Process {
  ...
  template<typename BindFn>
  inline auto then(BindFn&& k) &&;
}
```

Here a template type `BindFn` denotes a closure, which must take a value of the `Item` data type and then return a new `Process` computation. To represent the closure, the template parameter is used for performance.

The same idea could be expressed through the standard C++ function data type, but that would be much less efficient! This is not a real definition already:

```
template<typename Item, typename Impl>
class Process {
  ...
  // N.B. Inefficient! The real simulator uses another approach as stated above!
  template<typename ThenItem>
  inline Process<ThenItem> then(std::function<Process<ThenItem>(Item&&)>&& k) &&;
}
```

For instance, if we want to hold the current process for 15 modelling time units and then for 5 modelling time units after the first delay, then we can write:

```
hold_process(15.0)
  .then([](Unit&& unit) { return hold_process(5.0); })
```

This is already a combined computation that holds the current process for 20(= 15 + 5) modelling time units, when used in the simulation. It is important to understand that this computation does nothing unless we involve it in the simulation. Later we will return to this subject.

Also the closure must return the `Process` computation. This is why we need the mentioned `pure_process` function. If we have no ready computation then we create it.

As a demonstration, let us take a very abstract example that takes the current value within `Process` computation and increments it:

```
template<typename Impl>
inline auto inc_process(Process<int, Impl>&& m) {
  return std::move(m)
    .then([](int x) {
      return pure_process(x + 1);
    });
}
```

The resulting type is actually has type `Process<int, SomeImpl>`, where `SomeImpl` is inferred by the C++ compiler. If you remember, an object of this data type can be converted to `Process<int>` with the default data type for the second template parameter.

As usual, general concepts are difficult to understand and realise. This is indeed a general concept known as *monad*. For example, the experimental futures in C++ are monads. If you could apply the futures with help of combinators to create asynchronous programs, then you will be able to create and run models with help of DVCompute++ Simulator. Here I just want to note that DVCompute++ Simulator has a solid and powerful theoretical basis, which means that you can model many and many simulation activities.

5 Random Delay

There is a plenty of different functions for generating random values in DVCompute++ Simulator, but we need one of them, which will be used in the example later (the pseudo-code).

```
Process<double, auto> random_exponential_process(double mu);
```

It holds the current process for random interval distributed exponentially with the specified average value. This delay is a side effect of the computation. But the computation itself returns the value of time delay applied. This is what described by type `double` in the resulting `Process` computation.

Actually, this function just generates a random value for the delay interval and then combines it with the described above `hold_process` function that already implements the delay. In other words, this is a compound `Process` computation. Soon we will see how we can create composite computations to model some useful activity.

There is yet another function that performs the same side effect, but it ignores the final result:

```
Process<Unit, auto> random_exponential_process_(double mu);
```

6 Machine Process

To illustrate the concepts introduced above, we will consider a model of the machine that works during some up-time distributed exponentially. Then the machine breaks down and the repair-person repairs the machine during the time interval, which is distributed exponentially too. After the machine is repaired, it continues working. Please note that this is a recursively defined behaviour.

```
const double up_time_mean = 1.0;
const double repair_time_mean = 0.5;

Process<Unit> machine_process() {
  return random_exponential_process(up_time_mean)
    .then([](double up_time) {
      return random_exponential_process(repair_time_mean)
        .then([](double repair_time) {
          return machine_process();
        });
    });
}
```

To combine different parts together, we use the `then` combinator described before. Each continuation proceeds only after the preceding part of the computation is finished. Here the sequential behaviour is described.

The modelling time changes in steps, but the code itself looks like it might be linear. Moreover, the code is defined recursively as an infinite loop, but it is actually stopped right after the simulation terminates, when approaching the final time.

Just calling the `machine_process` computation does not run the discontinuous process. As it was noted before, the resulting computation must be involved yet in the simulation to be launched. Soon we will see how it can be done.

Finally, in the real model we could add counters to gather statistics about up-time and repair-time. They can be implemented with help of special references as it will be described later.

7 Event Handlers

The introduced `Process` computation is not alone in DVCompute++ Simulator. There are other simulation computations too.

The `Process` computation representing a discontinuous process can be run within `Event` computation. The latter is often used for representing an event handler. It actually implies some action that occurs at the current modelling time. It cannot be interrupted. It cannot have time delays. It cannot be blocked and so on. This is just a function of time.

```
template<typename Item, typename Impl = internal::event::BoxedImpl<Item>>
class Event;

template<typename Impl>
Event<Unit, auto> run_process(Process<Unit, Impl>&& comp);
```

The `Event` computation can be used for defining models in terms of the *event-oriented paradigm* of discrete event simulation as we will see later, while the `Process` computation is an instrument of the *process-oriented paradigm*.

Each simulation computation has the corresponding boxed representation, where the `Impl` template parameter has the default data type. For example, the boxed representation of the `Process` computation is just `Process<Item>` for some `Item`. Similarly, the boxed representation of the `Event` computation is `Event<Item>` for some `Item` too. Each boxed computation contains data which are allocated in the dynamic memory, but the `Impl` parameter type is erased. A non-default `Impl` type usually indicates that the corresponding computation data are allocated in the stack of the computer, but not always.

The key feature of the `Event` computation is that it allows us to define the event handler that will be actuated at the specified time point unless the simulation terminates earlier.

```
template<typename Impl>
Event<Unit, auto> enqueue_event(double event_time, Event<Unit, Impl>&& event);
```

There are similar combinators like `then` for the `Event` computation that actually allow us to define a wide range of simulation activities. Only the time delay must be either represented with help of the just mentioned `enqueue_event` function, or must be represented in terms of the `Process` computation or more high-level `Block` computation that was not introduced yet.

Thus, the `Event` class is self-sufficient, but every `Process` computation must be run within the `Event` computation.

8 Simulation Run

If we take some `Event` computation and launch it in the start time, then we receive another computation that is called `Simulation`. The latter represents some action that occurs within simulation run. It is not bound up with the modelling time point already. It is related to the entire simulation.

```
template<typename Item, typename Impl = internal::simulation::BoxedImpl<Item>>
class Simulation;

template<typename Item, typename Impl>
class Event {
  ...

  Simulation<Item, auto> run_in_start_time() &&;
  Simulation<Item, auto> run_in_stop_time() &&;
}
```

While the first function runs the computation in start time, the last one runs it in the final time point of simulation.

This is not the lowest simulation computation yet, but this is already a place, where we can finally run our simulation of the machine which we wrote before.

9 Simulation Specs

Now it is time to define the simulation specs by which we will be able to run the simulation. The specs are defined by the following structure that defines the start time of simulation, the final time, some small time step and the random number generator type, respectively. The last two attributes require more attention.

```
Specs specs { 0.0, 1000.0, 0.1, GeneratorSpec() };
```

Please think of the time step as the integration time step if we applied Euler's method or the Runge-Kutta method, when integrating the set of differential equations. Actually, this attribute has namely these roots, but it can be used for other purposes, for example, for defining the grid step size, when collecting statistics for the deviation chart. So, the time step should be quite small but not less. The number of steps should have a quite reasonable value.

The random number generator can return a reproducible sequence of values for the sequential simulation if required, but the optimistic distributed simulation mode supports a non-reproducible pseudo-random number sequence only (in some cases you can define a reproducible generator yourself). The latter is specified by value `GeneratorSpec()`. It will generate a conditionally non-reproducible pseudo-random number sequence both for the sequential simulation mode and two supported distributed simulation modes (the optimistic and conservative ones).

Given the simulation specs, we can run the specified `Simulation` computation and receive the result.

```
template<typename Item, typename Impl>
class Simulation {
  ...
  Result<Item> run(const Specs* specs) &&;
}
```

This method is namely what launches the entire simulation!

Thus, the idea is to represent the whole simulation model as a `Simulation` computation, regardless of that how large is our model, and then call this simple function to start simulating.

Here the `Result` type indicates either a successful result, or failure. To get the result or throw an exception, we can use one of the following helper functions:

```
template<typename Item>
const Item& expect_result(const Result<Item>& result);

template<typename Item>
Item& expect_result(Result<Item>& result);

template<typename Item>
Item expect_result(Result<Item>&& result);
```

10 Running Machines

Earlier we defined the machine process. If we take two machines then we can launch them in the start time point. The resulting model will have type `Simulation<Unit, Impl>` for some `Impl`.

```
  auto comp = run_process(machine_process())
    .then([=](Unit&& unit) {
      return run_process(machine_process());
    })
    .run_in_start_time()
    .then([=](Unit&& unit) {
      return pure_event(Unit())
        .run_in_stop_time();
    });
```

Here we do need to run an empty computation in the final time point so that two discontinuous processes would continue working until the simulation terminates. Otherwise, all the activity would finish directly in the start time. But the computation launched in the stop time forces the event queue to process all event handlers. The discontinuous processes are implicitly transformed into such event handlers too.

The final step is to launch our simulation.

```
auto result = std::move(comp).run(&specs);
```

11 Collecting Statistics

There are two structures to represent statistics summary in DVCompute++ Simulator. The first structure is called `SamplingStats`. It is used for collecting *observations-based statistics*. For example, it can be a wait time in the queue or resource. The second structure `TimingStats` is used already for collecting the *time persistent variable statistics*. For instance, it can be the queue size statistics in the same resource.

The both structures represent immutable data types. The usual approach is to mutate some `Ref` reference within `Event` computation to modify the statistics value. It is quite efficient and it works both for the sequential and distributed simulation modes.

12 Queue, Facility, Storage and GPSS

DVCompute++ Simulator provides a `Facility` that may preempt discontinuous processes with less transact priorities. Actually, DVCompute++ Simulator supports an embedded domain-specific language, which is similar to the popular GPSS modelling language, but only models are defined in terms of the C++ programming language. This is based on the described above `Process` computation. Only to represent blocks, there is the `Block` computation, which can be run within the `Process` computation. Similarly, there are `Storage` and `Queue` counterparts from GPSS in DVCompute++ Simulator too.

13 Block Computation

The `Block` computation denotes a function that takes some `Input` and then returns the `Process<Output>` computation within some discontinuous process. In functional programming such a function is called the *Kleisli arrow*.

```
template<typename Input, typename Output,
  typename Impl = internal::block::BoxedImpl<Input, Output>>
class Block {
  ...
  template<typename ThenOutput, typename ThenImpl>
  Block<Input, ThenOutput, auto> then(Block<Output, ThenOutput, ThenImpl>&& next) &&;
}
```

If `a1` and `a2` are some blocks, then `std::move(a1).then(std::move(a2))` specifies a compound computation, where we initially process the input by the first computation `a1` within discontinuous process, receive the intermediate

result, which becomes an input for the second computation that also processes its input within its discontinuous process. The next result becomes the result of the compound computation.

This concept is very useful to specify the model in a manner very similar to the GPSS programming language. So, our machine process could be specified as a block computation as well.

```
static Block<SharedRef<Transact<int>>, Unit> machine_chain(const SharedRef<Queue>& queue) {
  return queue_block<int>(queue, 1)
    .then(advance_block<SharedRef<Transact<int>>>(random_exponential_process_(up_time_mean)))
    .then(depart_block<int>(queue, 1))
    .then(advance_block<SharedRef<Transact<int>>>(random_exponential_process_(repair_time_mean)))
    .then(delay_block([=]() {
      return machine_chain(queue);
    }));
}
```

Here the `SharedRef` type is similar to the standard `std::shared_ptr` smart pointer, but optimised for execution within simulation. The `Transact<int>` template class denotes the transact that has a value of the specified integer data type.

If the template parameters could be inferred by the C++ compiler, then we would write the same in a more short way (the pseudo-code):

```
static BlockChain machine_chain(const SharedRef<Queue>& queue) {
  return queue_block(queue, 1)
    .then(advance_block(random_exponential_process_(up_time_mean)))
    .then(depart_block(queue, 1))
    .then(advance_block(random_exponential_process_(repair_time_mean)))
    .then(delay_block([=]() {
      return machine_chain(queue);
    }));
}
```

This is an infinite block chain that models the same machine process, where we also gather statistics by using the `queue` instance.

Finally, we have to delay the last computation so that we could ever create an object representing the whole computation. We delay the recursive call within the corresponding closure.

14 Magic Error C1060 of Visual C++

Here we use so much of the code inlining that some compilers such as Visual C++ protests sometimes against it by generating the magic error C1060. Fortunately, there is a simple workaround for this case. We can convert the computation to its boxed representation even if it will slow down the performance.

```
static Block<SharedRef<Transact<int>>, Unit> machine_chain(const SharedRef<Queue>& queue) {
  return queue_block<int>(queue, 1)
    .then(advance_block<SharedRef<Transact<int>>>(random_exponential_process_(up_time_mean)))
    .then(depart_block<int>(queue, 1))

#if defined(_MSC_VER)
    // a workaround for the MSVC compiler error C1060
    .operator Block<SharedRef<Transact<int>>, SharedRef<Transact<int>>>()
#endif

    .then(advance_block<SharedRef<Transact<int>>>(random_exponential_process_(repair_time_mean)))
    .then(delay_block([=]() {
```

```
      return machine_chain(queue);
    }));
}
```

It helps the compiler to simplify the code, but the performance slightly degrades. Therefore, I applied the corresponding conversion only for Visual C++, while GCC and Clang could ignore it. Actually, Visual C++ should be able to compile the represented code, but the error indeed may arise in a case of slightly more long constructs.

15 Model Initialisation

The most difficult part with simulation computations is that they increase the level of nested functions, each time we use the `then` combinator for monads. Therefore, DVCompute++ Simulator contains a helper optional class `Model` that allows us to greatly simplify the very process of the model initialisation.

```
Model model;
```

We gather statistics for machines. Hence, we have to initialise the future `Queue` instance.

```
auto queue = model.new_queue(std::string("Queue"));
```

To run blocks, we need two objects that would be then converted to two transacts for representing the machines.

```
auto machine_stream {
  random_int_uniform_stream(0, 0).take(2)
};
```

Here zeros means that the stream generates an infinite amount of data in the start time, but then we take only two first items. So, the resulting stream has size two.

Now, to plan the start of machines, we can create the transacts by the specified stream of two items and then process the both transacts by the block chain introduced before.

```
auto machine_start {
  run_process(stream_generator_block<int>(std::move(machine_stream), 1)
      .run([=]() { return machine_chain(queue); }))
    .run_in_start_time()
};
```

Here the transacts will have priority 1, for example. The actual priority values have no any meaning for this specific model.

We only created the `Simulation<Unit>` computation. Now we add it to the helper model instance so that the simulation model would launch the machines during its initialisation.

```
model.emplace_action(std::move(machine_start));
```

Finally, we initialise the model within `Simulation<Unit>` and return some statistics in the final modelling time, for example, the average queue content value.

```
auto comp {
  std::move(model)
    .init_in_start_time()
    .then([=](Unit&& unit) {
      return queue_content_stats(queue)
        .map([](SamplingStats<int>&& x) { return x.mean; })
        .run_in_stop_time();
    })
};
```

Now we launch the entire simulation to get the corresponding statistics for two specified machines.

```
auto result = std::move(comp).run(&specs);
auto x = expect_result(result);
```

16 Signals

DVCompute++ Simulator extensively uses the `Observable` template class to implement signals internally, but you can use it in your models too. The `Event` and `Observable` computations can be used for modelling hardware such as digital integral circuits. The event handlers may have priorities, which can be used for synchronising the signals. Also, the signals can be delayed through the event queue, merged or split.

17 Optimistic Distributed Simulation

Everything described above is applicable both for sequential simulation and distributed simulation based on the optimistic Time Warp method. You can launch the simulation on your laptop, but you can also create a distributed model and then launch it on the super-computer with help of DVCompute++ Simulator. In the latter case the MPI protocol is used for efficient communication between *logical processes*.

The difference between the both modes is that you have to use different C++ name spaces, but the simulation computations and entities have the same names. Only in case of distributed simulation, these computations and entities often require that types and closures must be copyable, i.e. they must have the copy constructor, while the sequential mode of simulation uses almost exclusively the move semantics.

The copying operation is required, because the distributed simulation may have rollbacks. Therefore, computations cannot be irreversible. Every time the event queue has to launch the event handler, it copies the handler to create its clone. But usually, it is quite a cheap operation. Possibly, the need to clone is the most significant difference between the sequential mode and the optimistic distributed simulation mode in DVCompute++ Simulator. They are very similar in other aspects.

Also, in case of distributed simulation you create logical processes that can send messages to each other. The act of sending the message is expressed in terms of the `Event` computation, while the act of receiving messages is naturally expressed within `Observable` computation.

18 Conservative Distributed Simulation

DVCompute++ Simulator supports the conservative method of distributed simulation too. The same functions and the same simulation concepts work as before! Only we must specify so called the *lookahead* parameter, which is the lower bound for the minimal possible delay in modelling time units, which may arise when passing messages between the logical processes.

The both methods of distributed simulation are important. The both are supported by DVCompute++ Simulator.

19 Nested Simulation

Finally, there is another simulation mode supported by DVCompute++ Simulator, where all the simulation concepts introduced before remain actual. This is one of the versions of nested simulation, where we can create sub-models within any model or sub-model at any modelling time point, launch these sub-models in their future modelling time, receive some estimation and then return to the current model or sub-model to proceed with the execution from the current modelling time point and by using the received estimation. Such nested models usually have an exponential complexity. Therefore, the level of branching sub-models must be limited by some value.

This simulation mode is related to Theory of Games. It can be useful to find some optimal strategy of behaviour. In the author's opinion this is quite a very interesting field, which is not studied well yet.

20 Unified Simulation Approach

Nevertheless, all the considered modes are based on the same unified simulation approach. After you have a practice in the simplest mode among them, i.e. the sequential simulation mode, you can start creating distributed or nested simulation models. This can be enjoying!

21 License

DVCompute++ Simulator is free for non-commercial purposes, but to use the simulator in commercial projects, you have to purchase a commercial license by contacting me at david.sorokin@gmail.com.

References

[1] David Sorokin. Distributed simulation with Aivika (in Russian). *Journal of Applied Informatics*, No.4 (82), 2019.