The Discrete Event Simulation Framework

Karl Gemayel

This tutorial, if I can call it that, is not meant to be a detailed description of how to design or implement discrete event simulations. Rather, it provides a motivation behind the idea of why a *framework* for such simulations is useful, and what the main components of that framework are. It also relies on you having already (at the very least) skimmed through the provided sample code.

As such, I would recommend having the sample code near you while reading, and trace the concepts I mention (especially near the end) through the concrete example you have been provided with.

Let us begin...

1 A butcher, a baker, and a candlestick maker walk into a pub...

"We're in a bit of a quandary" says the butcher to the other two. "A small glass of sherry, please", says the refined candlestick maker to the barman as they sit down. "And two *regulars* for my friends here."

"I just don't have that kind of money," says the baker. "A hundred quid a month just to have bread sent out from my shop to my customers! And for what? A dime a dozen! I barely have any money left to put bread on the table." The candlestick maker eyes him conspicuously. He had always wondered whether it was his friend's diction that inspired his choice of occupation, or the other way around.

"At least you're doing something," interrupts the butcher, taking a glass from the barman who had just joined in. "I'm still waiting and hoping people just show up at the shop. I don't own a car. I haven't the faintest idea how to get my products out. I tried driving my mate's car once; it did not go well. I promised him a month's worth of pig's feet to keep him from pressing charges."

"Why the long faces?" intrudes the barman cheerfully. "Business is bad as usual," answers the candlestick maker gloomily. "Apparently, walking to a shop is too much of an inconvenience nowadays. We're trying to figure out how to get all our products sent off to our customers."

"Ah, then, it's your lucky night!" says the barman standing up, and he ushers one of his mates over from behind the bar. "This is my friend Tom," he says, "and he's got just what you need: A *delivery* truck!". "A *what* truck?" asks the butcher. "*Delivery*" replies the barman. "*D* for daphnomancy, *E* for eisegesis, *L* for lizard, ..."

"Oh, delivery." interrupts the butcher, having vaguely heard of this new fad. "What's that then?" "Well," says Tom, "my job is to take items from one person and give them to another, for a fee of course." "Really?" says the candlestick maker, putting down his sherry. "Can you do candlesticks then?" "Sure," replies Tom. "I can do just about anything as long as it's packaged correctly."

"Not bread, though!" inquires the baker. "Yes." says Tom. "Yes?" says the baker incredulously. "Yes? Candlesticks and bread? In the *same place*? You're mental, you are!" "As long as it's *packaged* properly, I'd deliver my own brother if I had to. All I require is that I have no responsibility to know or care about what's inside the package. I can guarantee that your items will be delivered safe and sound, no questions asked. All you have to do is package your items securely, and tell me where they need to be delivered."

"How much does it cost?" asks the butcher. "Due to my ability to deliver different types of items at the same time, I can keep my costs down. So, on average, 60 quid a month for all three of you together. That's 20 each." "That's incredible!" says the butcher. He looks at his friends. "What do you say? Are you in?" "Yes!" exclaims the baker. The candlestick maker takes a slow sip from his glass. He then pulls a 20 out of his pocket and places it on the table. "First

month's service." he says insipidly. "I'll expect you at 7 in the morning," and he stands up and walks towards the exit. On his way out, he overhears the baker say "Talk about a sourdough." He smiles to himself and leaves the pub.

2 Love thy framework

The above illustrates the main idea behind the discrete event simulation (DES) framework.

The End

... Is it not clear? Fine, I'll explain further. The butcher, the baker, and the candlestick maker produce three quite distinct products, and they are looking for a way to deliver these products to a customer. Well, delivering products can depend on the nature of the products. What the entrepreneurial Tom did was the following: he designed a system whereby if an item is properly packaged, then he can deliver it to its final destination, *no matter* what the item is.

In DES, we have an application that needs to be modeled as a sequence of (discretely spaced) events. The application can represent an airport runway, a cake-baking service, or even a chemical process. Events for these applications range from an aircraft landing on a runway, to ovens preheating to 375° F, to adenine binding to thymine.

That said, what's common between all the above is that they all depend on *events* being executed in some defined order. Well, let's invite Tom over and have him offer his services again, tailored to our DES formulation. "I can guarantee that your *events* will be *delivered* back to you in the order prescribed by you, no questions asked. All you have to do is *package* your items securely, and tell me *when* they need to be delivered."

In other words, as long as the application can package its events properly, Tom will take that event and give it back at the time requested by the application.

2.1 DES Framework

We start off with the little-known 11th commandment:

Thou shalt not concern an Engine with the details of an Application.

Loosely translated, that means: The Simulation Engine should be completely *independent* of the Application. If, like Tom, we want to provide a service

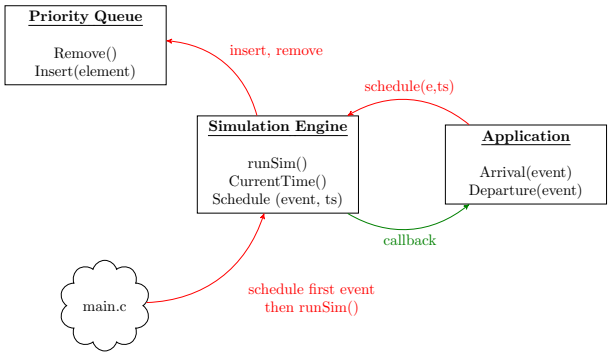

Figure 1: The Discrete Event Simulation Framework

(Simulation Engine) that is generic enough to accommodate any type of request (Application), then our design of the service should be completely independent of the details of any particular request. Fortunately, in DES, this is not hard to do. Since all our applications effectively operate at the level of Events, we can abstract away from the details of these events that pertain to any one application. In fact, all the simulation engine needs to do is "hold on" to an event for a while, until that event's time-stamp has been reached. After that, it just needs to hand it back to the application.

Let's look at a visual representation of the design of a DES (Fig 1). The main idea is that Application will *package* events and send them to the Simulation Engine to hold by using the `schedule(event, timestamp)` function. Then, when the simulation time equals `timestamp`, the Simulation Engine hands this event back to the Application through what's referred to as the `callback` function. There are many ways to implement callbacks. One common way is to define a function prototype (say, `EventHandler`) in the Simulation Engine, but put the implementation of that function in the Application. Re-read that sentence carefully and make sure you know what I'm talking about.

The reason why the prototype is placed in Simulation Engine is so that the engine has access to that function (and can call it). The reason why the implementation is in Application is simply due to the 11th commandment; i.e. that the simulation engine does not, can not, and should not know the details of the application. In other words, the specifics of how that event is to be executed should only be privy to the Application.

2.2 Technically Speaking

Now that the idea behind the DES framework is clear, implementing it in C would require us to dwell in the deep and empty *voids* of the language.

The reason is that C is a (statically) typed language; meaning, every variable should have a predefined type. So how is the simulation engine expected to "hold" an event item if it cannot know what the type of that event actually is? We need a "type" that can hold (or, more specifically, *point to*) anything. This is the `void*` pointer. Let's see how it works.

Suppose an airport runway application creates an event of type `AirEventData` (where `AirEventData` is defined in the Application); i.e. `AirEventData* e = malloc(sizeof(AirEventData));` This can then be passed on to the simulation engine by "packaging" it in a `void*`; specifically, the schedule function would have the form `schedule(void* event, int ts)` where `ts` is the time-stamp of when the event should be executed.

Accordingly, the application would call `schedule(e, ts)`, and the simulation engine will be given the memory address of the event `e` to hold securely, until the simulation time equals `ts`, at which point it should give it back to the application.

The key thing here is that the simulation engine only knows what the memory address is, but *nothing else*; i.e. it does not know the type of element stored at that location (since it uses `void*`), nor does it know the size of the data that's stored there. Fortunately, it doesn't *need* to know, since all it has to do is hand back that memory address to the application at the correct time.

Which brings us to the final step of the puzzle. How is it handed back? Well, remember that `EventHandler` function I spoke about earlier? The one declared in the Simulation Engine but implemented in the Application? Yes, that one. Well it's signature looks something like this: `void EventHandler(void *e)`. So at the correct "time", the simulation engine will just call this function and pass as parameter the item it picked up earlier from the application.

Finally, the magical unravelling happens in the implementation of `EventHandler` (which occurs in Application). The application *knows* that what it originally sent off to the engine had the type `AirEventData`. And it made a pact with the engine that guarantees that the engine will return the item *as is*. In other words, the Application is confident enough that what the engine returns, although packaged in a `void*`, is actually of type `AirEventData*`. Therefore, all it has to do is cast it back; i.e. `AirEventData* d = (AirEventData*) e;` and, *tada*! The event that had been previously created by the application has passed through the simulation engine, and is now back safely in the hands of the application. *Thanks, Tom.*

3 Conclusion

This tutorial has likely not contributed to your detailed understanding of how discrete event simulations are designed and implemented. My apologies. What I hope it did do, however, is portray the clearly distinct roles of the Application and Simulation Engine. In particular, and I'll leave you with this, you should never, *ever* attempt to introduce elements of the application into the simulation engine. *Never*. I mean it.