CSLP Project

Hamish Hutchings

21/12/2015

Introduction

The Computer Science Large Practical (*cslp*) task was to define and write a piece of software (in *C*) that would perform simulations and experiments on a bus simulation network in a city. I never managed to get it to a completely working state, yet I have documented and explained my decisions throughout the process, and will lay out my current working implementation below.

Continuity Decisions

When deciding how I would construct the state of my program, I looked into the classical mutating state construct. I initially thought that it would be fairly easily modeled using the concept of a single State struct that modifies itself on every `tick` (where a `tick` is a single state-transition).
After some thought, I considered a possible modification to the linked list idea. I would generate a new state based on the previous state, then hold a reference to the previous state in the now current state. This would give me the advantage of every state being accessible, as well as the ability to do post-run analysis. Another major advantage would be that I wouldn't have to worry about previous states accidentally interfering with current information. As every successive state is explicitly created, and only explicit copying from the previous state to the current one is allowed. Although this is considerably less space-efficient, I never came anywhere near any significant amount of memory. Running several times gives us around ~400k bytes of memory being allocated (per run).
One potential problem with the model I decided to implement is that multiple events happening at the same time was not supported (A potential solution being to increase granularity of events, or move to continuous time).

Event states

After some deliberation, I decided that the best way to manage different events in a (non-Object Oriented) language was to have some form of 'information' stored in the state that will let me pick the relevant event-struct that contains the information required for the event that occurs at this instance. This is simply implemented using a *switch/case* statement. I broke down the required events into several structs. The events are listed below:

```
#define NULL_EVENT                    0
#define PASSENGER_SUBSCRIPTION_EVENT 1
#define PASSENGER_EMBARK_EVENT        2
#define PASSENGER_DISEMBARK_EVENT     3
#define BUS_ARRIVAL_EVENT             4
#define BUS_DEPARTURE_EVENT           5
```

you will notice that there is a `NULL_EVENT`. I included a `NULL_EVENT` for non-categorized events. There were several other events that needed to be included into this list, namely things such as `PASSENGER_BEGIN_DISEMBARK` and `PASSENGER_END_DISEMBARK`. These are not events that need outputting to the event-stream, but need to be unique and distinguishable. The advantage of having all interactions with modification of state is that the modeling of the system becomes a much easier task, (the handling and listing of events, known to me as event-driven programming (flux)).

Libraries Used

I went through several testing libraries before settling on *CuTest*. I found it easy to include and carry around. I also found it very easy to use. It is a very simple testing framework, and doesn't allow much in terms of testing options, but I found it more than adequate, and found I could quite easily read the source to understand how to use it.
Earlier on in the project, I used an in-memory graph-database called *WhiteDb*, thinking that this would allow me to easily construct and work with the network. Unfortunately due to my recent adoption of *C*, I didn't feel like the time investment would pay itself off.

Data-Structure

I tried to implement a data-structure that I could pass a reference around the program of, such that I could build up relationships between entities quickly and easily. I found it incredibly useful for example to always have a reference to my

config struct accessible at `state->config`, regardless of where I was. In this way, I never had to worry about storing and passing around the dimensions of the map, as it was easily accessible. Another point being, as I never mutate a struct after creating it, I guaranteed myself (only through convention) that the values in the config file would be correct. I attempted to have several references to the same stops around my *state* struct, but never managed to fully understand how to achieve this. This is why I had several integer values that directly mapped to stops, as this was easier to implement without deeper understanding in *C*.
With the sharing of data-structures, (and using this method of non-mutation), we can safely reference structs without fear of the data we are referencing changing and becoming unpredictable.

Stop Network

Building the stop network took a couple of attempts. I initially thought I could somehow define a 2d-array and define each element in the array as an edge between the two vertices. After considering this for a little while, I felt like this would not allow scope for space optimization, and would make it harder to directly find all neighbouring vertices of a vertex. also, without wrapping the data in an abstraction, it would be considerably harder to modify and evolve the implementation to something more optimal.
I went instead for something resembling a form of hash implementation. Currently I have implemented a static array, but the implementation I use allows for using of linked-lists to improve space-efficiency. The implementation was done by creating an array of edges in each stop, and filling the array with all outgoing edges for that stop. In this way, it would be easy to walk through nodes in an iterative fashion and allow for space optimization.

Testing

Testing was implemented using the *CuTest* library. The test-suite was split into several suites that focused on different aspects of the program. The initial test-file was designed around testing the config file was properly populated when reading the file from disk.
I attempted to write a series of checks/unit-tests (bundled into small test-units) for every logical unit of work that the project performed. Anything that could return content, or modify data that will directly affect the running of the program is tested. I tested formatting of output, and correct construction of structs in initial states, correct creation of successor states and correct final states (i.e. when the application reaches end-conditions).

Conclusion

Although I didn't manage to fulfill all the requirements of this project, I feel like the solution I was working towards was a unique and interesting implementation. I have implemented a functional-like approach to the handling of data to preserve the integrity of references to data-structures. In addition, in the event-loop all actions were dispatched in an event-like system allowing predictable logical paths of execution. Overall, although the requirements were not met, this was an interesting and challenging project which expanded my understanding of *C* programming techniques.