Computer Science Large Practical

Introduction

This is the program for 2012 Computer Science Large Practical as set out in the Handout

This file is written using Markdown. While you can convert it to html using Markdown.pl or any other converter it is integrated with doxygen and appears as the main page of the doxygen documentation and also bundled as a pdf

Building

The program requires GNUstep and was built on Ubunut 12.10 and Arch Linux.

Ensure you have run GNUstep.sh to setup your environmental variables. See the GNUstep documentation for more details

cd into the root of the Cslp directory and run `make`. This will build the binary and write it to `Cslp.app/Cslp`.

Building Documentation

Documentation for the project can be built using `make docs`. This requires:

- gimli
- sed and `rm` (Obviously should be installed)
- Doxygen 1.8 or greater
- Graphviz
- md2man

Due to these requirements, the documentation has been prebuilt

(Note: There are a lot of warnings when building the doxygen documentation. This is because I didn't and don't intend to document everything. For example in most cases it should be clear what `-(void)dealloc;` does and documentation adds nothing).

Documentation

Other than this document there is the man page. As this program lacks an installer (The current development system is not the same as the system that will mark it and packages are a different format) the man page is viewable via `man -l doc/CSLP.1`

Documentation for the code (generated using doxygen) can be accessed from the codedocs.html file found in the doc directory.

Running the Application

The Unix way of running the program is to pass the program the input script through stdin and let the program write the output to stdout. For example:

```
cat ./examples/decay.txt | ./Cslp.app/Cslp
```

Due to the requirements of the application it is also possible to pass as the first argument the path to the file to read. For example:

```
./Cslp.app/Cslp exampes/decay.txt
```

If you want to output elsewhere than stdout then the following are equivalent:

```
./Cslp.app/Cslp exampes/decay.txt > output.txt
./Cslp.app/Cslp exampes/decay.txt --output output.txt
```

Command Line Arguments

Run `./Cslp.app/Cslp --help` to see an up to date list of command line options

Configuration Script

The configuration script is the same as described in the Handout, but with the following differences.

- Identifiers must be consistent but you can use words as opposed to letters (so you are not limited to 25 molecules)
- t has an alias time, so both `t` and `time` are reserved identifiers
- Comments can be written on the same line as settings
- Support for numbers in scientific notation

- Validation is preformed on the scripts. It should be impossible to use an invalid script and warnings are given for scripts that are have problems (it is possible to treat warnings as errors using the `--wall` command line option)

Output Formats

There are two things that can effect the output, both configurable via the command line. The first is the aggregator (`--aggregator`) which can group up simulator state changes. Valid options for this are:

- `PassthroughAggregator` - All state changes are logged
- `HundredMsAggregator` - At least a hundred ms needs to have passed for a state change to be logged
- `ExactHundredMsAggregator` - Writes the state exactly every hundred milliseconds. Possibly slightly slower than the other aggregator as it has to do more allocations
- `ResultOnlyAggregator` Writes the last output only

The second option that can effect the output is the writer itself (`--writer`). Valid options are

- `AssignmentCsvWriter` - Writes in the (invalid) Csv format specified by the assignment
- `RfcCsvWriter` - Csv writer that conforms to RFC 4180

Logging

It was suggested that I add logging. While I am not too fussed (I am fairly sure about the correctness of the program given the number of unit tests) I suppose it is a good idea to have.

As I didn't want to add huge amounts of requirements for something that would be marked (i.e. I didn't want the marker to have to spend 4+ hours setting up an environment to build the application) I wrote a very basic logging library that does what I need.

Logging can be enabled using command line parameters see Command Line Arguments for more details.

Note: Early startup errors such as incorrect command line parameters can only be logged to stderr as at this point no logs have been created.

Running Tests

From the project root directory run `gnustep-tests`. There are tests for most parts of the system.

The code quality of the tests isn't terribly high as there is quite a lot of *almost* duplicated code, this is in part due to the verboseness of Objective C in that it takes a huge number of lines to setup the state. The other issue was that I couldn't find a decent unit testing and object mocking framework that worked reliably with gnustep (despite having spent several hours trying to set some of them up). This meant that I didn't have features such as test auto discovery and methods that were automatically run before each tests.

Performance

The biggest performance improvement that can be done is to use the ResultOnlyAggregator aggregator as the slowest part of the system is writing the output (It isn't just the IO, string concatenation is also slow). Also if you want performance, avoid writing the results to stdout as it isn't buffered so much slower than writing to a file.

There are several performance enhancements to the simulation that work well with larger systems. They can be enabled via `--sdm`, `--ldm` and `--dependency-graph`. These adversely effect performance with smaller systems.

- SDM (Simple Direct Method) slowly sorts the list of reactions by their rate of recent occurrence.
- LDM (Logarithmic Direct Method) preforms a binary search against the list of reactions rather than a linear search. This makes SDM useless but will probably only have a noticeable impact on performance for very large numbers of reactions
- Dependency Graph maintains a dependency graph of what applying a reaction will effect. It uses this to minimize the number of calculations needed.

I did not build a big enough system to find where one where using logarithmic search made a significant performance gain and I would suspect that with a larger number of reactions this would start to make a difference.

Although (I would assume) outside the scope of this project would be to write the file as binary, which would avoid the string concatenation problems (but the data would be in a less useful format). It would also be possible (on a multicore machine) to write on one thread and do the simulation on another thread, although care would have to be taken with the cost of synchronization overhead.

While it is (compared to the performance problem of the output) fairly minor, it would be advantageous to write the Simulator in C. This is because when using Objective C it is hard to avoid heap allocations in the simulator main loop as the Objective C container types (NSArray, NSSet etc) cannot store primitive types so have to be wrapped (for example having to create a NSValue to do a pointer lookup). Given that the assignment wanted the code to be Objective C, I didn't experiment with writing a version of the simulator in C.

Other code

tests/Testing.h is not my code and comes from gnustep. It is a set of macros that provide helpers to test functionality (such as exceptions being raised). It is included as it doesn't seem to live in a GNUStep include directory.

Coding Style

Where possible I have tried to stick to a idiomatic Objective C coding style. There is one noticeable area where I haven't done so and that is with pointer variable declarations. It seems that the standard way of doing things is to declare them like so (note the position of the asterisk):

```
MyClass *myVariable;
```

This is a follow on from C, which for some reason that when declaring several variables of the same type on the same line, you must do so like this (note the second asterisk).

```
MyClass *myVariable, *myOtherVariable
```

However, it is my opinion that the `*` is part of the type, not of the name and so should live with the type information such as:

```
MyClass* myVariable;
```

I simply avoid declaring the multiple variables of the same type on the same line.

Commenting

My general view on code commenting is that if you have to comment the code *implementation* to explain what you did or why you did it then the code isn't good enough. Rather than adding a comment, it is better to fix the code. There are of course a few exceptions such as:

- Optimized Code
- Code working around bugs/problems

There is also the problem with comments that in effect it leads to maintaining two implementations. One being the comments and the other being the code itself. Even with the best of intentions comments will drift out of sync with the code and when that happens it leads to confusion (Which is correct? The comments or the code?).

This partly stems out of having worked on a large project (averaging 8 developers) with Skyscanner that had almost no comments, and which tended to mean that developers couldn't write messy code and "fix" it by adding a comment and the functions tended to be cleaner. E.g. You couldn't get away with writing:

```
int doFoo(int time);
```

As it isn't clear what it returns and you know that it takes a time, but is it hours? decades? So you would would write:

```
TimeSpan* doFoo(TimeSpan* time)
```

Which adds type safety and removes ambiguity from the arguments without having to resort to comments

While I haven't gone to that extreme in this project, where I have thought that comments may be needed in the implementation I have instead re-factored the code to attempt to make it more readable. I also tried to avoid using primitive types where possible and instead used objects, for example TimeSpan.