The Tracing package
a more in depth explanation

DEVS Ex Machina

April 25, 2015

This document contains a more in depth explanation of the inner workings of the tracing package. We decided not to add this directly to the report to keep the report clean and concise.
This document is not supposed to be a manual, but rather an explanation of the techniques used in this package.

1 package structure

The entire tracing package consists roughly of three parts. The first part is is the `Tracers` class. This class contains a set of tracers and provides a layer of abstraction for managing these tracers. The second part are the individual tracers. Each tracer generates specific output for the events of the simulator. The final part deals with this output. It makes sure that the output is generated in the correct order.

Tracers class

2 Class Structure

The class structure can be best described as a recursive inheritance tree. Through the use of variadic template parameters[1], each superclass takes care

[1] variadic template parameters: Wikipedia, cplusplus.com

of one template parameter and inherits from the class that will take care of the rest.
The base case is a class that does not contain a tracer object.
Because the recursive case has more specific template parameters, it gets precedence over the base case.

```
TracersTemplated<A, A, B> test;
```

The test object now contains 2 tracer objects with type `A` and one tracer object of type `B`.
The complete inheritance tree would look like this:

```
TracersTtemplated<> //base case
        △
        |
TracersTemplated<B>
        △
        |
TracersTemplated<A, B>
        △
        |
TracersTemplated<A, A, B>
```

The base case defines the operations and values that `TracersTemplated` should contain regardless of how many tracers are used.
The other classes only add one new tracer object that they own privately.

3 Methods

Certain methods deserve special attention because of the way they are implemented.

3.1 Constructor

The non-default constructor can be used to initialize all the tracers at once. It uses the copy constructor to initialize the tracer members. There is also a default constructor.

3.2 getParent

The private `getParent` method returns a reference to the parent class part of the object, effectively granting access to the interface provided by that type. It works by casting the `this` pointer to the appropriate pointer type and dereferencing it. The use of `static_cast` over a c-style cast assures type correctness.

example To continue the example from earlier, `TracersTemplated<A, A, B>` would return the reference to the `TracersTemplated<A, B>` part of the object.

3.3 getByID

This function allows direct access to the n^{th} registered tracer.
Standard C++ doesn't allow function member template specialization for a non-specialized class template[2], therefore we used a workaround.

3.3.1 sfinae

The implementation relies heavily on sfinae - Substitution Failure Is Not An Error[3].
Basically, you present the compiler a few options for implementations of a particular function template. Only one of those options should compile correctly with any given template arguments. When the compiler fails to substitute the template arguments in one option, it continues with the other options until it either finds a single suitable option. When no suitable option can be found, the result is a compiler error.

3.3.2 implementation

The implementation offers three very similar implementations of the `getTracer` function:

```
//option 1
template<std::size_t n>
```

[2]specialization rules: cppreference.com
[3]sfinae: Wikipedia

```
typename std::enable_if<n <= sizeof...(TracerElems) && n != 0,
  typename std::tuple_element<n,
    std::tuple<T, TracerElems...>>::type>::type&
getByID()
{/*...*/}

//option 2
template<std::size_t n>
typename std::enable_if<n==0 ,
  typename std::tuple_element<n,
    std::tuple<T, TracerElems...>>::type>::type&
getByID()
{/*...*/}

//option 3
template<std::size_t n>
typename std::enable_if<sizeof...(TracerElems) < n, void*>::type
getByID() = delete;
```

The `std::tuple_element<n, std::tuple<T, TracerElems...>>::type` denotes the type of the tracer that should be returned and is not really relevant to this explanation.
The `std::enable_if<Condition, type>` struct defines a typedef `type` if the condition evaluates to true. If the condition evaluates to false, the struct is empty.
Therefore, depending on the value of the template argument `n`, the return value of one of the three options is defined and the others are not, resulting in a substitution error.
If n is too large, that is, n is larger than or equal to the amount of used tracers, the last option is taken. Because that function is deleted, the compiler[4] generates an error that points to where the function was invoked.
Note that you cannot use the `>` operator in the `std::enable_if` condition. The compiler gets confused because it fails to distinguish between the comparison operator and the closing angle bracket of the template.

[4]Only tested with GCC 4.9.2.

4 What about std::tuple?

`std::tuple<...>` already offers a lot of functionality that we implemented ourself. The reason why we chose to reinvent the wheel over using `std::tuple<...>` is because our implementation makes it very easy to loop over all the registered tracer objects and call one of their functions.
Looping over a `std::tuple<...>`, while possible[5], involves a lot of boilerplate code for a simple function call.

Individual tracer

A Tracer has to generate output for certain events and send it to some output sink.
Each individual tracer class has the exact same structure. Policy-based programming[6] is used to separate the output writing logic from the output creation logic. This allows the user to use their own output method without having to rewrite all the existing tracers.

4.1 Policies

An output policy dictates where the generated output will go to. At the moment, we have implemented a policy for writing the output to `std::cout` and another policy for writing the output to a file.

5 Planned changes

At the moment, we have only implemented one tracer, the verbose tracer. The other tracers (XML, JSON and CELL) will have a very similar structure. We could just create a single base class and use virtual functions to provide the implementation of the tracer-specific functionality. However, this would defeat the purpose of custom designing the template classes we have already. Another option is to use the Curiously Recurring Template Pattern (CRTP)[7].

[5] Looping `std::tuple<>`: Stack Overflow
[6] Policy-based programming: Wikipedia
[7] CRTP: Wikipedia

Curiously Recurring Template Pattern This pattern allows a base class to call functionality of a derived class without having to use virtual functions or function overloading. It is in effect a way to deal with this kind of polymorphic behavior at compile time. The compiler knows exactly which functions will be called by the base class and can therefore use optimization techniques that are not always available when using runtime polymorphism.

Output Scheduling

In order to make sure that the output generation is consistent in between runs and doesn't contain any output that is reverted, the tracers don't immediately send the generated output to their output sink.

6 Trace Message

The tracers put their generated output in a message, much like the messages that the models can send to eachother. They contain the following information:

- *timestamp* The timestamp of the message dictates the ordering of the messages.
- *execute function* When a trace message is handled, this function is executed. This function should send the output generated by the tracer to the output sink.
- *coreID* Each simulation has a unique core ID. When this core requests a rollback, we can selectively purge only the trace messages that belong to this core. Otherwise, if one core does not participate in the rollback, for example when it did not receive an antimessage, its output is not touched.
- *delete function* In some cases, memory could be allocated on the heap. This function can be used to free that memory. It will be called when the message is destructed, regardless of whether its *execute function* was called.

Both the *execute function* and the *delete function* are objects of type `std::function<void()>`. You can use `std::bind<...>(...)` to create a delegate.

7 Message handling

Unlike in the Python PDEVS implementation, trace messages are not scheduled together with the other messages, but rather in a separate scheduler. The following free functions are defined for managing the messages

- `void scheduleMessage(t_tracemessageptr message);`
  Schedule a single message.
- `void traceUntil(n_network::t_timestamp time);`
  Execute all trace messages with a timestamp less than the provided parameter.
- `void revertTo(n_network::t_timestamp time, std::size_t coreID = std::numeric_`
  Purges all trace messages from a core that have a timestamp greater than the timestamp `time`. The default value of the second parameter will indicate that the messages of *all* cores must be removed. The execute function of the trace messages will not be called.
  Note Removing the messages of a single core is more costly than removing the messages of all cores.
- `void clearAll();`
  Clears all remaining trace messages. Their execute function will not be called.
- `void waitForTracer();`
  Waits until the tracing thread is finished.

In the background, *traceUntil* will create a new thread that will perform the actual output. Therefore, rest of the simulation does not have to wait until this job is done. The current implementation can still be optimized further, for example by using a 1 reader/multiple writers approach. We will try to fix this for the beta version.