Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 247–255.

Implementation of a finite state machine with active libraries in C++*

Zoltán Juhász, Ádám Sipos

Department of Programming Languages and Compilers
Faculty of Informatics, Eötvös Loránd University
e-mail: {cad, shp}@inf.elte.hu

Abstract

Generative programming is an approach to generating customized programming components or systems. C++ template metaprogramming is a generative programming style. With *metaprogramming* we can reduce the runtime cost of our programs, more extensible libraries can be made, and the creation of *active libraries* is also supported. Active libraries are not passive collections of routines or objects, as are traditional libraries, but take an active role in generating code. Active libraries provide higher abstractions and can optimize those abstractions themselves.

Our goal is to demonstrate the connection between *Finite State Machines* and active libraries. One of the fields where Finite State Machines are applicable is the implementation of complex protocol definitions. Since often only the results of test cases of a protocol are obtainable, the developer himself has to define and implement his own state machine description. With the help of active libraries we are able to use a compile-time Finite State Machine implementation, and do some monitoring at compile-time, such as consistency checking of the state machine's state table, or error checking. Another aim of the compile-time state machine implementation is the enhanced effectiveness.

In this paper we introduce the Finite State Machine's formal definition, and then discuss the possible implementation techniques. We analyze the functionality, and place special emphasis on compile-time solutions. We describe the algorithms that carry out error checking and transformations on the Finite State Machine's state table. Our implementation of the state minimization algorithm based on the *Boost::MPL library* is also described.

Keywords: C++, Boost::MPL, Boost::Statechart, Finite State Machine, Template, Metaprogramming, Active Library

MSC: 68N19 Other programming techniques

*Supported by the Hungarian Ministry of Education under Grant FKFP0018/2002.

1. Templates and metaprograms in C++

Templates are one of the most important components of the expressiveness of C++. They were introduced into the language for supporting code reuse and to introduce a higher level of abstraction. Most C++ programs use data structures e.g. stacks, arrays etc. The language does necessitate that these containers must have an appropriate implementation for every type they will be used with, which is inconvenient since you have to write separate implementation for every new type introduced.

In C++ you can create constructs that can merge the common parts of the type-dependent code into a template class or function. These can be parameterized with an arbitrary type that meets every requirement needed by the template.

The benefits are obvious: the fact, that you need to write the common operations and algorithms for several types just once reduces the possibility of introducing flaws and supports code tracking.

In 1994, Erwin Unruh, a member of the C++ standardization committee presented a program, which was able to generate prime numbers during the compilation [1]. This was the first known template metaprogram. Later researches have pointed out that the meta-language defined by C++ templates is Turing-complete [2], so we are able to write loops and conditional statements in metaprograms.

2. Metaprograms and Active Libraries

In line with the evolution of programming languages, new and more developed libraries have emerged. Even FORTRAN programs utilized libraries that implemented the frequently occurring algorithms. With the growing popularity of Object-Oriented programming languages, libraries transformed and evolved: instead of a set of methods and functions, they contain a set of classes and inheritance hierarchies. However these libraries are still passive: the author has to make all the relevant decisions at the time of the implementation of the library. In some cases this is inconvenient.

Consider the following semi-C++ code [3]:

```
template <class T, class S>
? max( T a, S b)
{
    if ( a > b )
        return a;
    else
        return b;
}
```

We are writing a function that holds two parameters and returns the greater one. The two parameters' type can be different. It is clear, that we cannot decide

whether `a` or `b` will be bigger at runtime, so we cannot decide which type would be the right one for type of the return value. On the other hand with the help of metaprograms, we can develop a very simple strategy that tells us which type, `T` or `S` has a bigger representation field, so in a very plain situation we can select the suitable return type. For an average passive library writer this is a mission impossible, he cannot even make this simple call since does not know the actual type of parameter `a` or `b`$\cdots$ but the compiler does!

An *Active Library* takes an active role in the compilation [4]. They provide domain-specific syntax; they make decisions by the information provided by the compiler at compile time. They make optimizations, transformations and safety checking.

The aim of this paper is to highlight the connection between Finite State Machines and active libraries, and to introduce a special kind of active library, which implements the Moore reduction procedure [5] for a compile time defined Deterministic Finite Automaton.

3. Finite State Machine

The *Finite State Machine (FSM)* is a model of behavior composed of a finite number of states, transitions between those states, and optionally actions. It works over a set of states, the transitions between those states are managed by the transition function depending on then input symbol (event). In the rest of this paper we use the expression Finite State Machine (FSM), automaton or machine in terms of Deterministic Finite State Machine (DFSM).

3.1. A Mathematical model of Finite State Machine

A *transducer* Finite State Machine is a six tuple [6], consisting of

- A finite, non empty set of *input symbols*, let Σ denote it. We are referring to this set as the set of events.
- A finite, non empty set of *output symbols*, let Γ denote it.
- A finite set of *States* let S denote it.
- *Start or Initial state*, an element of let S, let $q_0 \in Q$ denote it.
- A *Transition function*: $\delta : Q \times \Sigma \to Q$.
- An *Output function*, let ω denote it.

In our model we are using a sixth component, which is a set of actions. The sixth component holds the available actions that can be executed through a transition between two states. Note that our model uses the Moore machine [6].

Deterministic Finite State Machines, Deterministic Finite Tree Automatons [12] etc., are a widespread model for implementing a communication protocol, a program drive control flow or lexical analyzer among others.

4. Common implementation techniques

There are a number of different FSM implementation styles from hand-crafted to professional hybrid accomplishments. In the next section we review some common implementation techniques.

4.1. The Hand-crafted

This is the simplest, maybe the most efficient but the least flexible solution of the implementation of a DFSM. The transition function's rules are enforced via switch-case statements. States and events are regularly represented by enumerations, actions are plain function calls. The biggest drawback of this implementation is that it is suitable only for the representation of simple machines, since the larger the automaton, the more error prone and hard to read its code.

4.2. Object-oriented

The object-oriented representation is a very widespread implementation model. The transition function behavior is modeled by the state transition table (STT). Table 1 shows a sample STT:

Current State	Event	Next State	Action

Table 1: State Transition Table

A good example of such an automaton implementation is the OpenDiameter Library's FSM [7].

One of the main advantages of the Object-Oriented solution over the hand-crafted version is that the state transition rules and the code of execution are separated and it supports the incrementality development paradigm in software engineering[13].

The drawback of an average OO FSM implementation is that you define and build your state transition table at runtime. This is definitely not free of charge. Sanity checking also results in runtime overhead.

4.3. Hybrid technique

The most promising solution is using the Object-Oriented and Generative - (metaprogramming) techniques side by side. States, events and even actions are represented by classes and function objects. And the most important thing is that the STT is defined at compilation time.

As soon as the STT is defined at compile time you can execute algorithms on it, you can transform it and after all you can optimize and check the whole state transition table.

An outstanding example of such a DFSM implementation is the Boost::Statechart Library [8] that is UML compatible, supports multi-threading, type safe and can do some basic compile time consistency checking.

5. Our implementation

Our goal was to develop a prototype of a library that carries out compound examinations and transformation on the state transition table and shows the relationship between Finite State Machines and Active Libraries over a template metaprogram implementation of the Moore reduction procedure. However, the complete review of our library is beyond the scope of this paper.

5.1. Applied techniques

We used many C++ template facilities extensively, such as SFINAE, template specialization, parameter deduction etc. [9]. In a metaprogram you use compile time constants, types instead of variables, objects respectively; template classes and functions instead of functions and methods. To simulate cycles and if-else statements we use partial and full template specializations.

Assignment is also unknown in the world of metaprograms, we use typedef specifiers to introduce new type aliases, that hold the required result.

We have already mentioned that C++ templates are Turing complete, we also saw that we are able to write loops and if-else statements, but we still need containers to write complex algorithms. We used *Boost::MPL*[10], which provides C++ STL-style[11] compile-time containers and algorithms.

5.2. Applied algorithms

In our model the State Transition Table defines a directed graph. We implemented the Moore reduction procedure, we used the Breadth-First Search (BFS) to isolate the graph's main strongly connected component and with the help of a special "Error" state we made it complete.

5.3. The State Transition Table

Much like the Boost::Statechart's STT, in our implementation states and events are represented by classes and structs (types). The STT's implementation based on the Boost::MPL::List compile-time container, as it follows:

6. Implementation of the algorithm

In the following we present the minimization algorithm implemented in our active library.

```
typedef mpl::list<
//    Current state   Event    Next state         Action
//    +-----------+----------+-------------+----------------------+
trans < Stopped   ,   play   ,   Playing   ,   &p::start_playback >,
trans < Playing   ,   stop   ,   Stopped   ,   &p::stop_playback  >
//    +-----------+----------+-------------+----------------------+
>::type sample_transition_table; // end of transition table
```

Figure 1: Implementation of our State Transition Table

6.1. Locating strongly connected components

The first algorithm that will be executed before the Moore reduction procedure is the localization of the strongly connected component of the STT's graph from a given vertex. We have used the Breadth-First Search to determine the strongly connected components, as it follows (some lines have been removed):

```
// Bread-First Search
template < typename Tlist, typename Tstate, typename Treached,
//                  STT ^    Start state ^   Reached states ^
           typename Tresult = typename mpl::clear<Tlist>::type,
//                  ^ Result list is initialized with empty list
           bool is_empty =  mpl::empty<Treached>::value  >
struct bfs
{
   // Proccessing the first element of the reached list
   typedef typename mpl::front<Treached>::type process_trans;
   typedef typename process_transition::to_state_t next_state;

   // (...) Removing first element
   typedef typename mpl::pop_front<Treached>::type tmp_reached_list;

   // (...) Merging reached and result list for further checking
   typedef typename merge2lists<tmp_result_list, tmp_reached_list>
         ::result_list tmp_check_list;

   // (...)  Recursively instantiating bfs template class
   typedef typename bfs< Tlist, next_state, reached_list,
         tmp_result_list, mpl::empty<reached_list>::value
                     >::result_list result_list;
};
```

During the implementation of the algorithm presented, we had to implement various processes that carry out various transformations on a Boost::MPL::List container. For example the `merge2lists`.

6.2. Making the STT's graph complete

The second algorithm that will be executed before the Moore reduction procedure is making the graph complete. We introduce a special "Error" state, which will be the destination for every undefined state-event pair. We test every state and event and if we find an undefined event for a state, we add the following row to the State Transition Table:

```
//    Current state   Event    Next state          Action
//    +-----------+----------+-------------+----------------------+
trans <     Stop  ,   pause  ,    Error    ,   &p::handle_error   >
```

Figure 2: Adding new transition

The destination state is the "Error" state. We can also define an error-handler function object[11].

The result after the previously executed two steps is a strongly connected, complete graph. Now we are able to introduce the Moore reduction procedure.

6.3. The Moore reduction procedure

Most of the algorithms and methods used by the reduction procedure have been already implemented in the previously introduced two steps. Quote from wikipedia:

"The concept is to start assuming that every state may be able to combine with every other state, and then separate distinguishable states into separate groups called equivalence partitions. When no more equivalence partitions contain distinguishable states, the states remaining in the same group as other states are an be combined. Equivalence partitions are numbered by the number of steps it took to get to that point. The 0th partition contains all the states in one group, the 1st partition contains states grouped by their outputs only. Every partition from then on has groupings that are based on which group from the previous partition those states' next state fell under. The procedure is complete when partition n is the same as partition $n + 1$." [5]

We have simulated partitions and groups with Boost::MPL's compile time type lists. Every partition's groups are represented by lists in lists. The outer list represents the current partition, the inner lists represent the groups.

Within two steps we mark group elements that need to be reallocated. These elements will be reallocated before the next step into a new group (currently list).

After the previous three steps the result is a reduced, complete FSM whose STT has only one strongly connected component. All of these algorithms are executed at compile time, so after the compilation we are working with a minimized state machine.

7. Results

The aim of the previously introduced techniques is to prove that we are able to do sanity checks and transformations on an arbitrary FSM State Transition Table. With the help of these utilities we can increase our automaton's efficiency and reliability without any runtime cost. We can also help the developer since warnings and errors can be emitted to indicate STT's inconsistency or unwanted redundancy.

Our algorithms are platform independent because we are only using standard facilities, defined in the C++ 2003 language standard (ISO/IEC 14882)[9].

There is only a little overhead on the code size and compilation time. The code size increased about 30 bytes; the compilation time about 2 second (with 50 states and randomly generated state transition rules).

8. Conclusion

In our paper we have presented an active library that can carry out checking and transformations on a FSM's State Transition Table. The algorithms were implemented by C++ template metaprograms and executed at compilation time, so these operations do not cause any runtime overhead; what is more, if reduction is possible, the FSM is expected to be faster during its execution. On the other hand with the aim of compile time checking and the emitted warnings and error messages the program code will be more robust, since the program can only be compiled if it meets the developer's requirements. These requirements can be assembled through compile time checking.

In this paper we have introduced the Active Libraries and presented an example of the relation between such libraries and finite state machines. We have also presented our active library implementation of the Moore reduction procedure and other algorithms, which is based on the Boost::MPL Library and has a strong connection to Boost::Statechart Library. The usage of such compile time algorithms has little impact on the code size and the compile-time.

References

[1] Unruh, E., Prime Number Computation, *ANSI X3J16-94-007/ISO WG21-462* (1994).

[2] Veldhuizen, Todd, L., C++ Templates are Turing Complete, (2003), `http://ubiety.uwaterloo.ca/~tveldhui/papers/2003/turing.pdf`

[3] Prokoláb, Z., *Advanced C++ Lessons*, `http://aszt.inf.elte.hu/~gsd/halado_cpp/`

[4] Czarnecki, K., Eisenecker, U. W., Gluck, R., Vandevoorde, D., Veldhuizen, T. L., Generative Programming and Active Libraries, *Springer-Verlag* (2000).

[5] Moore, E. F., *Moore Reduction Procedure*, Wikipedia, `http://en.wikipedia.org/wiki/Moore\_reduction\_procedure`

[6] Finite State Machine, Wikipedia, `http://en.wikipedia.org/wiki/Finite\_state\_machine`

[7] Fajardo, V., Ohba, Y., Open Diameter, `http://www.opendiameter.org/`

[8] Dönni, A. H., Boost::Statechart, `http://boost-sandbox.sourceforge.net/libs/statechart/doc/index.html`

[9] Programming languages C++, *ISO/IEC 14882*, (2003).

[10] Boost Libraries, `http://www.boost.org`

[11] Stepanov, A. A., Lee, M., *The Standard Template Library*, X3J16/94-0095, WG21/N0482 (1994).

[12] Novitzká, V., Mihályi D., Slodičák V., *Tree Automata in the Mathematical Theory*, Proceedings of 5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics, Poprad, ISBN 978-963-7154-56-0 (2007), 129–135,

[13] Szabó, Cs., Samuelis, L., *Notes on the role of the incrementality in software engineering*, Studia Universitatis Babes-Bolyai Informatica, vol. 51, no. 2 (2006), 11–18.

Zoltán Juhász, Ádám Sipos
H-1117 Budapest
Pázmány Péter sétány 1/C.
Hungary