Implementation of a Finite State Machine with Active Libraries in C++*

Zoltán Juhász, Ádám Sipos, and Zoltán Porkoláb

Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
e-mail: {cad, shp, gsd}@inf.elte.hu
H-1117 Budapest, Pázmány Péter sétány 1/C.

Abstract. Active libraries are code parts playing an active role during compilation. In C++ active libraries are implemented with the help of template metaprogramming (TMP) techniques. In this paper we present an active library designed as an implementation tool for Finite state machines. With the help of various TMP constructs, our active library carries out compile-time actions like optimizations via state-minimalization, and more sophisticated error-detection steps. Our library provides extended functionality to the Boost::Statechart library, the popular FSM implementation of the Boost library. We describe the implementation and analyze the efficiency.

1 Introduction

Generative programming is one of today's popular programming paradigms. This paradigm is primarily used for generating customized programming components or systems. *C++ template metaprogramming* (TMP) is a generative programming style. TMP is based on the C++ *templates*. Templates are key language elements for the C++ programming language [25], and are essential for capturing commonalities of abstractions. A cleverly designed C++ code with templates is able to utilize the type-system of the language and force the compiler to execute a desired algorithm [31]. In template metaprogramming the program itself is running during compilation time. The output of this process is still checked by the compiler and run as an ordinary program.

Template metaprograming is proved to be a Turing-complete sublanguage of C++ [6]. We write metaprograms for various reasons, here we list some of them:

- *Expression templates* [32] replace runtime computations with compile-time activities to enhance runtime performance.
- *Static interface checking* increases the ability of the compile-time to check the requirements against template parameters, i.e. they form constraints on template parameters [18, 23].

* Supported by GVOP-3.2.2.-2004-07-0005/3.0

– *Language embedding* makes it possible to introduce domain-specific code into a C++ program via a metaprogramming framework. Examples include SQL embedding [11], and a type-safe XML framework [13].
– *Active libraries* [29]. act dynamically during compile-time, making decisions based on programming contexts and making optimizations. These 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.

Finite State Machines (FSMs) are well-known mathematical constructs, their practical applications include but are not limited to lexical analyzers, performance tests, and *protocol definition implementations*. Most protocols are described by a FSM, since FSMs provide a clear framework for distinguishing the possible state transitions when the protocol is in a certain state. However, 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.

FSMs play a central role in many modern software systems. Besides their functionality, their correctness and effectiveness is also crucial. Unfortunately, recent implementation techniques provide no support for features like detecting unreachable states and carrying out automatic state reductions. This lack of features may reduce the quality and the effectiveness of FSM code used in critical applications.

With the help of active libraries we are able to define state machines, do sanity checking on their state tables, and enhance their run-time effectiveness at compile-time in a fully automated manner. Such process can either detect consistency errors during the compilation process, or produce a correct and optimized FSM for run-time usage.

Our goal is to demonstrate the possibility to implement and effectively use active libraries matching the above criteria. Our library is capable of carrying out compile-time operations and performs various checkings and optimizations on a state machine.

The paper is organized as follows. In section 2 we discuss C++'s templates and template metaprogramming concepts. Section 3 introduces the Finite State Machine's formal definition. Section 4 describes common implementation techniques for finite state machines. We discuss the possible implementation techniques in section 5. The code efficiency and compilation time measurement results are presented in section 6. Future development directions and related work are discussed in section 7.

2 C++ Template metaprograms

2.1 Compile-time programming

Templates are an important part of the C++ language, by enabling data structures and algorithms to be parameterized by types. This abstraction is frequently

needed when using general algorithms like finding an element in a data structure, or data types like a *list* of elements. The mechanism behind a list containing integer numbers, or strings is essentially the same, it is only the *type* of the contained objects that differs. With templates we can express this abstraction, thus this *generic* language construct aids code reuse, and the introduction of higher abstraction levels. Let us consider the following code:

```
template <class T>
class list
{
public:
    list();
    void insert(const T& x);
    T first();
    void sort();
    //...
};
```

```
int main()
{
    ...
    list<int> li;  // instantiation
    li.insert(1928);
}
```

This *list* template has one type parameter, called *T*, referring to the future type whose objects the list will contain. In order to use a list with some specific type, an *instantiation* is needed. This process can be invoked either implicitly by the compiler when the new list is first referred, or explicitly by the programmer. During instantiation the template parameters are substituted with the concrete arguments. This newly generated code part is compiled, and inserted into the program.

The template mechanism of C++ is unique, as it enables the definition of *partial* and *full specializations*. Let us suppose that for some type (in our example `bool`) we would like to create a more efficient type-specific implementation of the *list* template. We may define the following specialization:

```
template<>
class list<bool>
{
    // a completely different implementation may appear here
};
```

The specialization and the original template only share the *name*. A specialization does not need to provide the same functionality, interface, or implementation as the original.

2.2 Metaprograms

In case the compiler deduces that in a certain expression a concrete instance of a template is needed, an implicit instantiation is carried out. Let us consider the following code demonstrating programs computing the factorial of some integer number by invoking a recursion:

```
// compile-time recursion
template <int N>
struct Factorial
{
    enum { value = N *
      Factorial <N-1>::value };
};

template<>
struct Factorial<1>
{
    enum { value = 1 };
};

int main()
{
    int r=Factorial<5>::value;
}
```

```
// runtime recursion
int Factorial(int N)
{
    if (N==1) return 1;
    return N*Factorial(N-1);
}

int main()
{
    int r=Factorial(5);
}
```

As the expression `Factorial<5>::value` must be evaluated in order to initialize `r` with a value, the `Factorial` template is instantiated with the argument `5`. Thus in the template the parameter `N` is substituted with `5` resulting in the expression `5 * Factorial<4>::value`. Note that `Factorial<5>`'s instantiation cannot be finished until `Factorial<4>` is instantiated, etc. This chain is called an *instantiation chain*. When `Factorial<1>::value` is accessed, instead of the original template, the full specialization is chosen by the compiler so the chain is stopped, and the instantiation of all types can be finished. This is a *template metaprogram*, a program run in compile-time, calculating the factorial of 5.

In our context the notion *template metaprogram* stands for the collection of templates, their instantiations, and specializations, whose purpose is to carry out operations in compile-time. Their expected behavior might be either emitting messages or generating special constructs for the runtime execution. Henceforth we will call a *runtime program* any kind of runnable code, including those which are the results of template metaprograms.

C++ template metaprogram actions are defined in the form of template definitions and are "executed" when the compiler instantiates them. Templates can refer to other templates, therefore their instantiation can instruct the compiler to execute other instantiations. This way we get an instantiation chain very similar to a call stack of a runtime program. Recursive instantiations are not only possible but regular in template metaprograms to model loops.

Conditional statements (and stopping recursion) are solved via specializations. Templates can be overloaded and the compiler has to choose the narrowest applicable template to instantiate. Subprograms in ordinary C++ programs can be used as data via function pointers or functor classes. Metaprograms are first class citizens in template metaprograms, as they can be passed as parameters to other metaprograms [6].

Data is expressed in runtime programs as variables, constant values, or literals. In metaprograms we use `static const` and enumeration values to store quantitative information. Results of computations during the execution of a metaprogram are stored either in new constants or enumerations. Furthermore, the execution of a metaprogram may trigger the creation of new types by the compiler. These types may hold information that influences the further execution of the metaprogram.

Complex data structures are also available for metaprograms. Recursive templates are able to store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favorite implementation forms of expression templates [32]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [16].

2.3 Active libraries

With the development of programming languages, user libraries also became more complex. *FORTRAN* programs already relied heavily on programming libraries implementing solutions for re-occuring tasks. With the emerging of object-oriented programming languages the libraries also transformed: the sets of functions were replaced by classes and inheritance hierarchies. However, these libraries are still *passive*: the writer of the library has to make substantial decisions about the types and algorithms at the time of the library's creation. In some cases this constraint is a disadvantage. Contrarily, an *active library* [29] acts dynamically, makes decisions in compile-time based on the calling context, chooses algorithms, and optimizes code. In C++ active libraries are often implemented with the help of template metaprogramming. Our compile-time FSM active library also utilizes TMP techniques.

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. The transitions between the 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). Deterministic Finite State Machines, Deterministic Finite Tree Automatons etc. are a widespread model for implementing a communication protocol, a program drive control flow or lexical analyzer among others. The solution of a complex problem with a FSM means the decomposition of the problem into smaller parts (states) whose tasks are precisely defined.

3.1 A Mathematical model of Finite State Machine

A *transducer* Finite State Machine is a six tuple [20], consisting of

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

The working mechanism of a FSM is as follows. First the FSM is in the Start state. Each input symbol (event) makes the FSM move into some state depending on the transition function, and the current state. If the event-state pair is not defined by the function, the event in that state is not legal. For practical purposes we introduce a 7th component to the FSM, which is a set of actions. These actions that are executed through a transition between two states. Note that our model uses the Moore machine [20].

4 Common implementation techniques

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

4.1 Procedural solution

This is the simplest, but the least flexible solution of the implementation of a DFSM. The transition function's rules are enforced via control structures, like 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 no framework for sanity checking is provided, therefore the larger the automaton, the more error prone and hard to read its code [22]. Such implementations are rare in modern industrial programs, but often used for educational or demonstrational purposes.

4.2 Object-oriented solution with a state transition table

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` contains a state of the automaton, `Event` is an event that can occur in that state, `Next State` is the following state of the state machine after the transition, and `Action` is a function pointer or a function object that is going to be executed during the state transition.

A good example of such an automaton implementation is the OpenDiameter communication protocol library's FSM [10]. One of the main advantages

Current State	Event	Next State	Action
Stopped	play	Playing	start_playback
Playing	stop	Stopped	stop_playback

Table 1. State Transition Table.

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. The drawback of an average OO FSM implementation is that the state transition table is defined and built at runtime. This is definitely not free of charge. Sanity checking also results in runtime overhead and incapable of preventing run-time errors.

4.3 Hybrid technique

The most promising solution is using the Object-Oriented and template-based generative programming techniques side by side. States, events and even actions are represented by classes and function objects, and the STT is defined at compilation time with the heavy use of C++ template techniques, like Curiously Recurring Template Pattern (CRTP) [8].

An outstanding example of such DFSM implementation is Boost::Statechart Library [9], which is UML compatible, supports multi-threading, type safe and can do some basic compile time consistency checking. However, Boost::Statechart is not based on template metaprograms, therefore it does not contain more complex operations, like FSM minimization.

5 Our solution

As soon as the STT is defined at compilation time, algorithms and transformations can be executed on it, and also optimizations and sanity checking of the whole state transition table can be done. Therefore we decided to step forward towards using template metaprograms to provide automatic operations at compile-time on the FSM. Our goal was to develop an inital study 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.

The library is based on a simplified version of *Boost::Statechart*'s State Transition Table. In the future we would like to evolve this code base to a library that can cooperate with *Boost::Statechart* library and function as an extension.

5.1 Applied data structures and algorithms

We used many C++ template facilities extensively, such as SFINAE, template specialization, parameter deduction etc[21]. In a metaprogram you use compile time constants and types instead of variables and objects respectively; class templates and function templates instead of funtions and methods. To simulate cycles and if-else statements we used recursive template instantiations and 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 used *Boost::MPL*[5], which provides C++ STL-style[19] compile-time containers and algorithms.

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

Much like the Boost::Statechart's STT, in our implementation states and events are represented by classes, structs or any built-in types. The STT's implementation based on the Boost::MPL::List compile-time container is described in Figure 1:

```
template< typename T, typename From, typename Event, typename To,
     bool (T::* transition_func)(Event const&)>
struct transition
{
    typedef T       fsm_t;
    typedef From    from_state_t;
    typedef Event   event_t;
    typedef To      to_state_t;

    typedef typename Event::base_t base_event_t;
    static bool do_transition(T& x, base_event_t const& e)
    {
        return (x.*transition_func)(static_cast<event_t const &>(e));
    }
};

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
```

Fig. 1. Implementation of our State Transition Table.

A transition table built at compile-time behaves similarly to a counterpart built in runtime. The field `transition_func` pointer to member function represents the tasks to be carried out when a state transition happens. The member function `do_transition()` is responsible for the iteration over the table. The state appearing in the first row is considered the starting state.

5.2 A Case Study

In this section we present a simple use case. Let us imagine that we want to implement a simple CD player, and the behavior is implemented by a state machine. The state transition table skeleton can be seen in Figure 2.

```
typedef mpl::list<
//    Current state   Event    Next state        Action
//    +-----------+----------+-------------+----------------------+
trans < Stopped   ,   play   ,   Playing   ,   &p::start_playback >,
trans < Playing   ,   stop   ,   Stopped   ,   &p::stop_playback  >,
trans < Playing   ,   pause  ,   Paused    ,   &p::pause          >,
trans < Paused    ,   resume ,   Playing   ,   &p::resume         >,
                     ... duplicated functionality ...
trans < Stopped   ,   play   ,   Running   ,   &p::start_running  >,
trans < Running   ,   stop   ,   Stopped   ,   &p::stop_running   >,
trans < Running   ,   pause  ,   Paused    ,   &p::pause          >,
trans < Paused    ,   resume ,   Playing   ,   &p::resume         >,
                        ... unreachable states ...
trans < Recording ,   pause  ,   Pause_rec ,   &p::pause_recording>,
trans < Paused_rec,   resume ,   Recording ,   &p::resume_rec     >
//    +-----------+----------+-------------+----------------------+
>::type sample_trans_table; // end of transition table
```

Fig. 2. Sample State Transition Table

The programmer first starts to implement the Stopped, Playing, and Paused states' related transitions. After implementing a huge amount of other transitions, eventually he forgets that a Playing state has already been added, so he adds it again under the name *Running*. This is an unnecessary redundancy, and in generaly could indicate an error or sign of a bad design. A few weeks later it turns out, that a recording functionality needs to be added, so the programmer adds the related transitions. Unfortunately, the programmer forgot to add a few transitions, so the Recording and Paused state cannot be reached. In general that also could indicate an error. On the other hand if the state transition table contains many unreacheable states, these appear in the program's memory footprint and can cause runtime overhead.

Our library can address these cases by emitting warnings, errors messages, or by eliminating unwanted redundancy and unreacheable states. The result table of the reduction algorithm can be seen here:

```
template struct fsm_algs::reduction< sample_trans_table >;
```

After this forced template instantiation, the `enhanced_table` typedef within this `struct` holds an optimized transition table is described in Figure 3:

```
typedef mpl::list<
//    Current state   Event    Next state         Action
//    +-----------+----------+-------------+----------------------+
trans < Stopped   ,   play   ,   Playing   ,   &p::start_playback >,
trans < Playing   ,   stop   ,   Stopped   ,   &p::stop_playback  >,
trans < Playing   ,   pause  ,   Paused    ,   &p::pause          >,
trans < Paused    ,   resume ,   Playing   ,   &p::resume         >,
         ... duplicated functionality has been removed ...
          ... unreachable states have been removed too ...
//    +-----------+----------+-------------+----------------------+
>::type sample_trans_table; // end of transition table
```

Fig. 3. Reduced Transition Table

5.3 Implementation of the algorithms

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

Locating strongly connected components The first algorithm executed before the Moore reduction procedure is the localization of the strongly connected component of the STT's graph from a given vertex. We use Breadth-First Search to determine the strongly connected components. After we have located the main strongly connected component from a given state, we can emit a warning / error message if there is more than one component (unreachable states exist) or we can simply delete them. The latter technique can be seen in Figure 4 (several lines of code have been removed):

Making the STT's graph complete The Moore reduction algorithm requires a complete STT graph, so 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 a new row to the State Transition Table. (Figure 5.)

The destination state is the "Error" state. We can also define an error-handler function object[19]. After this step, if the graph was not complete, we've introduced a lot of extra transitions. If they are not needed by the user of the state

```
// Breadth-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
{
   // Processing 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;

   // (...) Adding recently processed state table rows
   // to the already processed (reachead) list
   typedef typename merge2lists<tmp_result_list, tmp_reached_list>
         ::result_list tmp_check_list;

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

Fig. 4. Implementation of Breadth-First Search

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

The Moore reduction procedure Most of the algorithms and methods used by the reduction procedure have already been implemented in the previous two steps.

First we suppose that all states may be equivalent i.e. may be combined into every other state. Next we group non-equivalent states into different groups called equivalence partitions. When no equivalence partitions have states with different properties, states in the same group can be combined. We refer to equivalent partitions as sets of states having the same properties. [14]

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).

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

Fig. 5. Adding new transition.

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.

6 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 compile-time 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) [21], and elements of the highly portable Boost library. Supported and tested platforms are the following:

- Comeau C/C++ 4.2.45, 4.3.3
- Compaq C++ (Tru64 UNIX) 6.5
- GCC 3.2.2, 3.3.1, 3.4, 4.1.0
- Intel C++ 7.1, 8.0, 9.1
- Metrowerks CodeWarrior 4.2.45, 4.3.3
- Microsoft Visual C++ 7.1

In the following we present code size and compilation time measurements with the gcc 4.1.0 20060304 (Red Hat 4.1.0-3) compiler. The test consists of the definition and consistency checking of a state transition table.

6.1 Code size

The x axis represents the number of states, while y shows the resulting code size in bytes. At 0 states the program consisits of our library, while no state transition table is used. Binary code size is increased only when the first state is introduced. The graph shows no further code size increase when the FSM consists of more states. (Figure 6) The reason is that the representation of each state is a type, which is compile-time data. This data is not inserted into the final code.

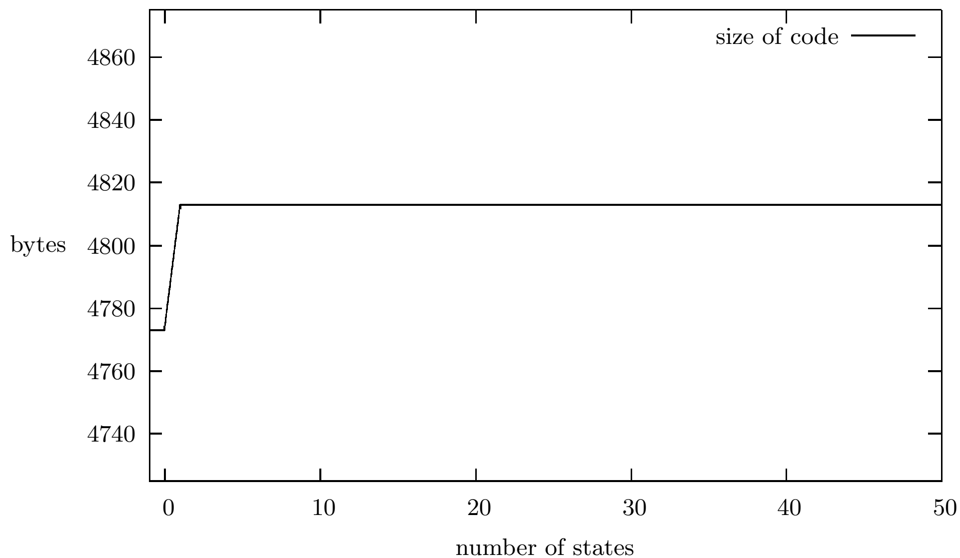

Fig. 6. Number of states and size of code

6.2 Compilation time

The testing method is essentially the same as above. Compilation time does not grow linearly with the introduction of new states. (Figure 7)

7 Related work and Future work

Final State Machine implementations vary from fully procedural [22] to object-oriented solutions [10]. Flexibility and maintanibility are becoming better, but the correctness of the created automaton ultimately depended on the programmers caution. Template techniques were introduced to enhance run-time performance [8], but not for providing sanity checks on the FSM.

The Boost Statechart Library supports a straightforward transformation of UML statecharts to executable C++ code [9]. The library is type safe, supports thread-safety and performs some basic compile time consistency checking. However, Boost::Statechart is not based on template metaprograms, therefore it does not contain more complex operations, like FSM minimization.

In the future we intend to extend the library with the following functionalities.

- *Warnings, error messages* - The library minimizes the graph without asking for confirmation from the programmer. Warnings and errors could be emitted by the compiler whenever an isolated node or reducible states are found.

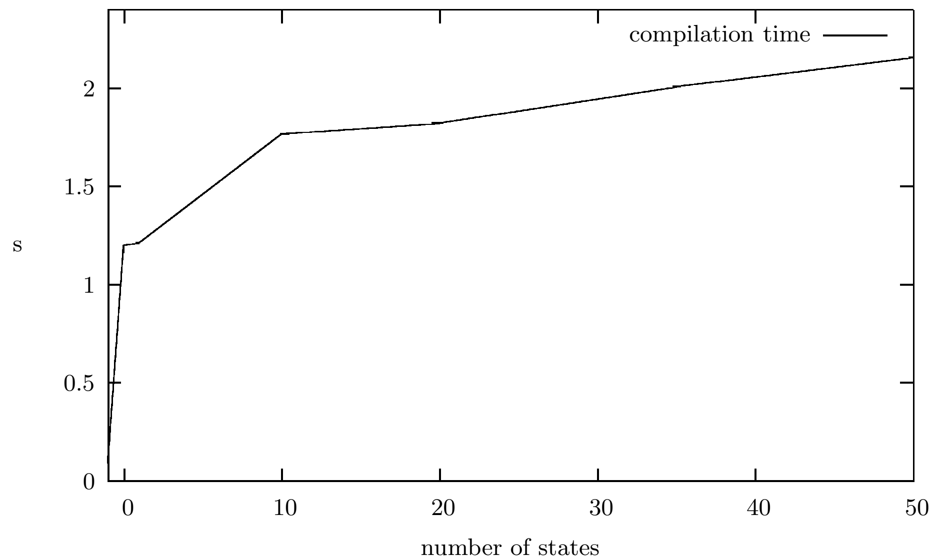

Fig. 7. Number of states and compilation time

– *Joker states, events* - In some cases it would be convenient to have a state in the FSM that acts the same regardless of the input event. Now we have to define all such state transitions in the STT. With future "joker" states and events, the STT definition would be simpler for the user, and also the reduction algorithm would have a smaller graph to work on. On the other hand the representation and the library logic would get more complex.
– *Composite states* - A composite state is a state containing a FSM. In case such state is reached by the outer machine, this inner automaton is activated. This FSM might block the outer automaton.

8 Conclusion

We created an active library to implement Final State Machines functionally equivalent to the Boost::Statechart library [9]. Our library is *active* in the sense that it carries out various algroritms at compile time. Algorithms include state machine reduction, and extended error checking.

The library carries out checking and transformations on a FSM's State Transition Table. The active library contains an implementation of the Moore reduction procedure and other algorithms. The algorithms are executed during compilation in the form of template metaprograms, therefore no runtime penalties occur. If the reduction is possible, the FSM is expected to be faster during its execution in runtime. The usage of such compile time algorithms has little impact on the code size.

On the other hand, with the aid of compile time checking and the emitted warnings and error messages the program code will be more reliable, since the program can only be compiled if it meets the developer's requirements. These requirements can be assembled through compile time checking.

Our implementation is based on the Boost::MPL and Boost::Statechart Libraries. As the library uses only standard C++ features, it is highly portable and successfully tested in different platforms.

References

1. David Abrahams, Aleksey Gurtovoy: C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, Boston, 2004.
2. Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
3. ANSI/ISO C++ Committee. Programming Languages – C++. ISO/IEC 14882:1998(E). American National Standards Institute, 1998.
4. Boost Concept Checking library.
   `http://www.boost.org/libs/concept_check/concept_check.html`
5. Boost Metaprogramming library.
   `http://www.boost.org/libs/mpl/doc/index.html`
6. Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)
7. Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glck, David Vandevoorde, Todd L. Veldhuizen: Generative Programmind and Active Libraries. Springer-Verlag, 2000
8. James O. Coplien: Curiously Recurring Template Patterns. C++ Report, February 1995.
9. A. H. Dnni: Boost::Statechart
   `http://boost-sandbox.sourceforge.net/libs/statechart/doc/index.html`
10. V. Fajardo, Y. Ohba, Open Diameter
    `http://www.opendiameter.org/`
11. Y. Gil, K. Lenz: Simple and Safe SQL Queries with C++ Templates. In Proceedings of the 6th international conference on Generative programming and component engineering, pp. 13-24, Salzburg, Austria, 2007
12. Douglas Gregor, Jaakko Jrvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. In Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'06), October 2006.
13. Y. Solodkyy, J. Järvi, E. Mlaih: Extending Type Systems in a Library — Type-safe XML processing in C++, Workshop of Library-Centric Software Design at OOPSLA'06, Portland Oregon, 2006
14. John. E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman: Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 2000
15. Zoltán Juhász: Implementing Finite State Automata with Active Libraries M.Sc. Thesis. Budapest, 2006.
16. Björn Karlsson: Beyond the C++ Standard Library, An Introduction to Boost. Addison-Wesley, 2005.
17. Donald E. Knuth: An Empirical Study of FORTRAN Programs. Software - Practice and Experience 1, 1971, pp. 105-133.

18. Brian McNamara, Yannis Smaragdakis: Static interfaces in C++. In First Workshop on C++ Template Metaprogramming, October 2000
19. David R. Musser, Alexander A. Stepanov: Algorithm-oriented Generic Libraries. Software-practice and experience, 27(7) July 1994, pp. 623-642.
20. J. E. Hopcroft, R. Motwani, J. Ullman: Introduction to Automata Theory, Languages, and Computation Addison-Wesley, 1969
21. Programming languages C++, *ISO/IEC 14882* (2003)
22. Miro Samek: Practical Statecharts in C/C++. CMP Books (2002)
23. Jeremy Siek and Andrew Lumsdaine: Concept checking: Binding parametric polymorphism in C++. In First Workshop on C++ Template Metaprogramming, October 2000
24. Jeremy Siek: A Language for Generic Programming. PhD thesis, Indiana University, August 2005.
25. Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)
26. Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley (1994)
27. Erwin Unruh: Prime number computation. ANSI X3J16-94-0075/ISO WG21-462.
28. David Vandevoorde, Nicolai M. Josuttis: C++ Templates: The Complete Guide. Addison-Wesley (2003)
29. Todd L. Veldhuizen and Dennis Gannon: Active libraries: Rethinking the roles of compilers and libraries. In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientic and Engineering Computing (OO'98). SIAM Press, 1998 pp. 21–23
30. Todd Veldhuizen: Five compilation models for C++ templates. In First Workshop on C++ Template Metaprogramming, October 2000
31. Todd Veldhuizen: Using C++ Template Metaprograms. C++ Report vol. 7, no. 4, 1995, pp. 36-43.
32. Todd Veldhuizen: Expression Templates. C++ Report vol. 7, no. 5, 1995, pp. 26-31.
33. István Zólyomi, Zoltán Porkoláb: Towards a template introspection library. LNCS Vol.3286 pp.266-282 2004.