Generating Functional Implementations of Finite State Automata in C# 3.0

Mihály Biczó

Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary

Krisztián Pócza

Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary

Abstract

Finite state automata (FSA) implementations are widely used in IT due to their rich application possibilities, flexibility, and their direct relationship with common goals set by regular business applications. Workflow-based programming has opened up a new exciting application area: workflows governed by FSA. The field is gaining more and more attention, the .NET 3.0 platform contains an engine called Workflow Foundation dedicated to managing workflows. However, due to the advantageous properties of FSA, it is often desirable to build applications around FSA implementations in simpler cases as well. In this paper an automata generator framework will be presented that makes it possible to use automata whenever possible. What makes this framework highly applicable and very flexible is the fact that the generation process might rely on not only static, but also dynamic information, so it can be performed during runtime as well. From the technical point of view, the interesting part is that the generated implementation is based on lambda expressions, the new functional enhancements of the C# 3.0 language.

Keywords: code generation, FSA, lambda expression, C# 3.0

1 Introduction

A finite state automaton [1] or state machine is a mathematical machine that, at a given time can be in exactly one state from a finite set of states. One important subset of states contains initial states. When the automaton is switched on, it can be in any of the initial states. When input from a finite alphabet is fed to the automaton, and certain conditions or constraints are met, the automaton can leave its current state and switch to a next state. This is called a state transition. If the automaton is in a final state when there is no more input, it is said to have accepted its input.

Finite state automata are popular because they can be applied to many practical problems. Nowadays, when workflow-based programming is gaining more and more importance among programming paradigms, a very interesting application area is opening up. Workflows can be mapped to an automaton that governs business rules driving the workflow [9]. Although sophisticated frameworks supporting workflow-based programming such as the Workflow Foundation engine of the .NET platform have emerged [2], yet there are cases when it can be important to bring implementation details closer to programmers. This should be done without exposing too many parts of the internal behavior to them or force them to setup complicated environments or even use additional designer tools.

In this paper we introduce a simple, yet effective way of integrating finite state automata to business oriented applications developed using C# 3.0 [3]. This has a positive effect of the manageability and testability of completed pieces of software. In order to achieve this aim, we are going to employ the new functional enhancements of the C# 3.0 language specification [4]. The functional programming paradigm [5] is a newcomer in the .NET world, however, its characteristics makes it a perfect candidate to support the implementation of FSA. Functional programming is said to be well applicable when we know exactly how we want to solve a given issue. When we talk about either business rules or finite state automata, the possible states and state transitions are specified along with the state transition conditions, this is how the automaton is described. All we have to do is to convert the formal description to functional rules, and the lambda expressions of C# 3.0 can accomplish this task. The conversion can take place automatically, so the implementation can be generated. The generation can happen prior to compilation, or, if necessary, even dynamically.

In this paper we show how functional implementations can be generated from declaratively defined automata in C# 3.0. In Section 2, we present the tools that have to be employed, anonymous delegates and lambda expressions.

In Section 3 we give a high level overview of the programming scenario when an automaton should be employed and list the required efforts on the part of programmers. We show configuration possibilities and also the concept of the generation process.

In order to verify the concept, in Section 4 we present a simple automaton generation process, and the usage of such an automaton. From the example the advantages of functional programming will be apparent: an easy-to read, and very compact code will be generated.

In Section 5 we summarize the results and give some hints how the existing implementation can be extended to feature more realistic scenarios.

2   Platform and tools

The .NET platform has seen a continuous and uncompromised evolution since the 1.0 version of the framework has been released by Microsoft in 2002. Although version 1.1 was a significant improvement over the previous release, it was ver-

sion 2.0 introduced in 2005 that contained important innovations at least on the programming languages side.

2.1 Anonymous methods

The biggest improvement of the C# 2.0 language was the generics construct. However, there were less important, but interesting new features as well. One of them was a new feature called anonymous methods. This feature allows programmers to create un-named or so-called anonymous methods. A similar concept of anonymous classes [6] has earlier been introduced in the Java programming language. Java does not define the notion of function pointers or delegates, it operates with interfaces and their implementation, that is why the anonymous level is that of the class.

Although the Java solution is more elegant in terms of object orientation, C# allows the usage of strongly typed function pointers or delegates.

```
List<string> names = new List<string>(new string[]
  { "Rudy", "Mike", "John", "Rebeca", "Melissa" });


public List<string> FilterRecordsWithDelegate()
{
  return names.FindAll(StartsWithR);
}


private bool StartsWithR(string name)
{
  return (name.StartsWith("R"));
}
```

Listing 1. Example with a simple delegate

In Listing 1, a method is defined to decide whether a string starts with 'R'. A list of strings is defined, and we pass our typed function pointer (the predicate) to the FindAll method of the generic List class.

In case of simple functions whose code is unlikely to be reused, it is inconvenient to define a separate function. C# 2.0 allows function pointers to be defined without naming them explicitly. The FilterRecordsWithDelegate function can be rewritten as follows using an anonymous delegate:

```
public List<string> FilterRecordsWithAnonymousDelegate()
{
  return names.FindAll(delegate(string name)
                      { return (name.StartsWith("R")); });
}
```

Listing 2. Previous example implemented with anonymous delegate

The StartsWithR function is defined in-place without assigning an explicit name

to it. Anonymous delegates are handy when a simple sort or find operation is implemented that does not deserve a dedicated function. However, seemingly they are against the readability and reusability of the code.

In C# 3.0, lambda expressions are a nice replacement of anonymous methods in terms of simplicity of syntax. Using a lambda expression the FilterRecordsWith-AnonymousDelegate method can be replaced with the following:

```
public List<string> FilterRecordsWithLambdaExpression()
{
  return names.FindAll(name => name.StartsWith("R"));
}
```

Listing 3. Example implemented using lambda expressions

The syntax resembles the syntax of a functional language [8]. In the first place, the simple functional-like syntax accounts for why we employ lambda expressions in our generator framework. In the above example the lambda expression returns a bool value that decides whether a string starts with ” R”. The really nice thing about lambda expressions is that they can be 'embedded' into each other which means that they can take lambda expression parameters and can return lambda expression. This embedding capability was the second motivation to represent the automaton as a chain of swiftly constructed lambda expressions. The details can be found in the next section.

3   High level overview

When constructing the automata generator framework, the key design points were the following:

(i) In order to easily describe an automaton from a definition, declarative syntax has to be used, in other words, it should be very easy for the programmer to construct the input of the generator framework. Declarative syntax means that the definition of the automaton has to be placed in a (possibly xml based) configuration file. The definition should be given using natural, low-level notions like states and state-transitions.

(ii) From each automaton description a separate class has to be constructed and placed in a single file. The file must be human readable so that it can be altered without running the generator again.

(iii) In order to satisfy flexibility requirements, the generation process should be possible to perform during runtime as well. If a new automaton description is obtained in runtime, a fully functional, strongly typed automaton should be possible to be used after the generation is completed.

(iv) The structure of different automata should be as 'close' to each other as possible.

In order to satisfy these conditions, the high-level operation of the framework is the following.

 (i) The configuration of an automaton is placed in a given location either before compilation time or in runtime. The schema of the configuration is part of the framework. The current schema supports one start state, one end state, and simple state transitions. This can be extended in future versions to handle more complex conditions when executing state transitions.

(ii) The configuration is verified against the given schema. If the verification fails, the generator framework aborts the generation process and warns the user.

(iii) An enumeration type is constructed for states, each state is a possible value of the enumeration type. The name of the enumeration type is StateValue.

(iv) An associative container (for example, a Dictionary) is constructed. The key of the associative container is a value of type StateValue (a given state), and the value associated with the key is a lambda expression whose input parameter is a possible input of the automaton, and the result is a StateValue (the next state).

In the current implementation, the declaration of the associative container looks like the following:

```
private Dictionary<StateValue, Func<string, StateValue>> transitions;
```

For each state, this container holds a lambda expression that decides the next state for a given input. The lambda expression is of the generic type Func<string, StateValue>. This denotes a lambda expression whose input is a string, and the result is a StateValue. (In the current implementation, the input of the automaton is always of type string. This can also be extended in future versions.)

 (v) Once we have the associative container for state transitions, the automaton (also a lambda expression) can be declared. A simple state transition is a lambda expression that maps a piece of input (a string) to a next state. If this idea is applied again, the whole automaton can be interpreted as a lambda expression that maps a sequence of inputs to a state. The current declaration of the automaton is the following:

```
private Func<List<string>, StateValue> automaton;
```

(vi) An automaton step lambda expression that places the state transitions to a context is generated. The signature of this lambda expression is the following:

```
private delegate StateValue AutomatonStepDelegate(List<string> input,
        StateValue currentstate, Dictionary<StateValue,
        Func<string, StateValue>> transitions,
        object transitionRule);
```

The whole automaton is represented as a lambda expression that maps a sequence of inputs to a lambda with the above mentioned signature.

(vii) A method called CreateTransitions is generated to the source file that fills the

transitions container based on values parsed from the configuration files.

(viii) The CreateAutomaton function is generated. The content of this method is constant (apart from the start and finish states) and looks like the following:

```
1. private void CreateAutomaton()
2. {
3.   AutomatonStepDelegate automatonStep = (A, C, D, E) =>
                              (A == null || A.Count == 0) ? C :
4.   (
5.   (C = (D[C])(A[0])) == StateValue.FINISH ? C :
6.   (E as AutomatonStepDelegate)(A.GetRange(1, A.Count-1), C, D, E)
7.   );
8.   automaton = (A) => automatonStep(A, StateValue.START,
                                      transitions, automatonStep);
9. }
```

Listing 4. Generated implementation of finite state automaton

In the body of the method the automatonStep lambda expression implements the behavior of the automaton. In the implementation A stands for input, C for the current state, D for the associative container containing state transition lambda expressions, and E for the description of the transition rules.

The generated code may look complicated at first, however, it is very simple. In line 3 the lambda expression checks whether the input sequence is null or empty. If this is the case, no state transition will occur, and the returned state will be the current state (C). In line 5 it selects the appropriate state transition from the associative container and applies it to the first element of the input sequence. If the resulting state is a finish state, it will be returned. Otherwise, the rule will be called recursively on the truncated input, the resulting state, and with the same transitions and transition management rules.

A last step of the function in line 8 is that the automaton is initialized with the whole input (A), the start state and the state transition container.

As a last step of the generation process the generated class is placed in a file that can later be used in the project. Using the automaton is trivial once the implementation has been generated:

```
public StateValue ProcessInput(List<string> input)
{
  return automaton(input);
}
```

Listing 5. Usage of the generated automaton

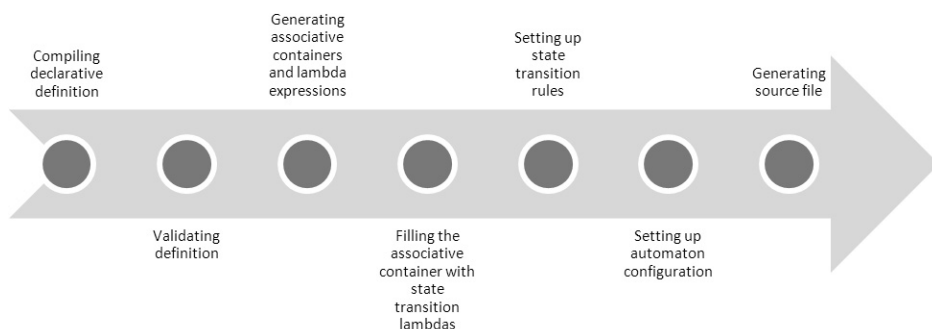In Figure 1, the main steps of the generation process are presented.

Figure 1. Main steps of automaton generation

In the following section a simple example using the generator framework will be described in detail.

4   Concept verification: a simple example

In this section a complete automaton generation process will be shown from the configuration to the generated code. The basic state transition rules of the automaton are borrowed from [7]. It is a SODA automaton. The SODA costs 20 cents, and the machine accepts nickels and dimes. The machine will not give change. Figure 2 represents the state transition diagram of the SODA machine:
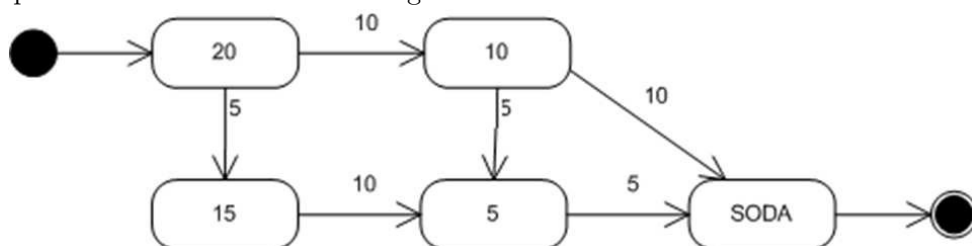

Figure 2. State transition of the SODA machine

One of the requirements of the generator framework was that the automaton should be described in a purely declarative way. Part of the declarative, xml based description of the above automaton is the following:

```
<?xml version="1.0" encoding="utf-8"?>
<AutomatonDescription
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        Name="SodaAutomaton"
        xmlns="wgt://elte.hu/ FiniteAutomatonSchema.xsd">
  <State Name="TWENTY" IsStartState="True" IsEndState="False">
    <StateTransition Input="10" ToState="TEN" />
    <StateTransition Input="5" ToState="FIFTEEN" />
  </State>
```

```
  <State Name="TEN" IsStartState="False" IsEndState="False">
    <StateTransition Input="5" ToState="FIVE" />
    <StateTransition Input="10" ToState="SODA" />
  </State>
  ...
</AutomatonDescription>
```

Listing 6. Declarative description of the SODA machine

Besides the states and the state transitions described by the input and the next state, we also mark start states and end states. The current implementation supports automata with one start state and one end state; however, this is not a strong requirement.

The next step is that the declarative description is validated according to a previously prescribed schema. The xml file is valid, so the constant container, StateValue enumerations and lambda expression definitions will be generated. In this case, the StateValue enumeration contains the TWENTY, FIFTEEN, TEN, FIVE, SODA values.

The next step is the generation of associative containers. For the automaton defined in Listing 6, the member function that fills the associative container would look like the following:

```
private void CreateTransitions()
{
  transitions=new Dictionary<StateValue, Func<string, StateValue>>();
  Func<string, StateValue> TWENTYTransition = (A) =>
    A == "10" ? StateValue.TEN : A == "5" ? StateValue.FIFTEEN :
                                              StateValue.TWENTY;
  transitions.Add(StateValue.TWENTY, TWENTYTransition);
  //...
}
```

Listing 7. Filling the associative container of states and state transitions

The next step is the setup of state transition rules, as described in Listing 4. The only thing that has to be accomplished is the replacement of start and finish states.

All remaining steps of automaton generation are definition-independent, so they can be performed even without the file that contains the declarative definition.

5   Summary

The popularity of workflow-based programming indicates that there has been a rediscovery of the applicability of FSA when solving practical problems. However, workflow-based programming often necessitates the usage of additional components or knowledge that is not suitable when developing smaller applications.

In this paper we have presented a simple framework that aims at generating the functional implementation of finite state automata. The benefits of our framework are the following:

(i) It uses easy-to understand, declarative syntax to describe an automaton, which is the most natural way (can be generated as well from other types of definitions)

(ii) C# 3.0 lambda expressions are employed and generated
  (a) They replace anonymous delegates of C# 2.0 and are syntactically simpler
  (b) Their syntax resembles that of a functional language
  (c) They provide an easy-to-read, yet very compact representation, the main part of the generated code is static;

(iii) It does not require additional components and third-party libraries.

The integration of the functional programming paradigm into an imperative, object oriented environment is an interesting experiment. The original concept was to support the implementation of simple, not reusable code fragments such as validation or comparison algorithms.

However, it turned out that they can be used to implement more complex structures such as simple automata as well.

Our pilot solution is limited in functionality, and there is a number of issues to work on. One interesting question is the handling of tail recursion. Unlike many functional environments, C# does not handle tail recursion. This might lead to stack overflows. However, there is a supported IL instruction called tail. Using this instruction it is possible to correct the emitted code at IL level.

We believe that this implementation can be extended in a number of ways. In the future, we plan to provide support for multiple start and end states as well as state transitions controlled by a guard condition. Seemingly, these extensions can be incorporated into the existing solution by applying only minor changes. However, implementing a non-deterministic automaton using lambdas is probably much harder.

A realistic automaton should react while consuming input. The current implementation does not contain this extension, but incorporation should not be hard since the reaction can easily be described as a delegate.

References

[1] Carroll, J., Long, D., *Theory of Finite Automata with an Introduction to Formal Languages*, Prentice Hall, Englewood Cliffs, 1989.

[2] Bruce Bukovics *Pro WF: Windows Workflow in .NET 3.0 (Expert's Voice in .Net)*

[3] Jesse Liberty, *Programming C# 3.0*, O'Reilly, ISBN 0-596-52743-8, 2007

[4] http://msdn2.microsoft.com/en-us/vcsharp/aa336745.aspx

[5] David R. Naugler, *Delegates and functional programming in C#*, ACM International Conference Proceeding Series; Vol. 61, pp. 17-27, 2004

[6] Richard G. Baldwin, *The Essence of OOP using Java, Anonymous Classes Java Programming Notes # 1640*

[7] http://web.cs.wpi.edu/čs1102/a05/Labs/lab2.html

[8] Simon Thompson, *Haskell: The Craft of Functional Programming (2nd Edition)*, Addison-Wesley, ISBN 0-201-34275-8, 1999

[9] http://www.objectmentor.com/resources/articles/umlfsm.pdf