On the Implementation of Finite State Machines

Jilles van Gurp & Jan Bosch
[Jilles.van.Gurp|Jan.Bosch]@ipd.hk-r.se
University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science
Soft Center, S-372 25 Ronneby
http://www.ipd.hk-r.se/[jvg|bosch]
tel +46 457-28651/fax +46 457-27125

Keywords. Finite State Machines, State pattern, Implementation issues, Blackbox frameworks

Abstract. *Finite State Machines (FSM) provide a powerful way to describe dynamic behavior of systems and components. However, the implementation of FSMs in OO languages, often suffers from maintenance problems. The State pattern described in [5] that is commonly used to implement FSMs in OO languages, also suffers from these problems. To address this issue we present an alternative approach. In addition to that a blackbox framework is presented that implements this approach. In addition to that a tool is presented that automates the configuration of our framework. The tool effectively enables developers to create FSMs from a specification.*

1 Introduction

Finite State Machines are used to describe reactive systems [7]. A common example of such a system is a communication protocol. TCP (Transmission Control Protocol [16]) for instance can be represented by a FSM. When such systems are implemented, the implementation reflects the finite State machine. FSMs are also used in OO modeling methods such as UML and OMT. Over the past few years, the need for executable specifications has increased [2]. The traditional way of implementing FSMs does not match the FSM paradigm very much, however.

Finite State machines can be described as a set of:

- *States*. The different states the FSM can be in.
- *Input events*. This could for instance be the arrival of some data at a FSM or it could be a Timer that runs out. It could also be the entry into a new State (after a state-change) or the exit of a State (before a State change). Input events are sometimes referred to as messages.
- *Output events*. This is an action that takes place after an input event arrives at a State machine. It could be part of a transition (see below) or it could be part of a State (for instance a state-exit-output event triggered by an input event that causes a transition.
- *Transitions*. A transition has a source State and a target state. Transitions are triggered by an input-event. The triggering may also cause the launching of one or more output-events.

In figure 1, the FSM-graph of our running example is shown (the example will be presented in section 2). The arrows are labeled with an event and an (optional) action/output event. A transition occurs if the system is in the State on the beginning of the corresponding arrow and the event on the transition occurs. The action is executed and afterwards the system is in the State on the end of the arrow. This simple model can be extended in many ways. An obvious extension is to make events conditional. An other possible extension is to have State entry and State exit actions that are executed when a State change occurs.

In this paper the following definition of a State machine will be used: A State machine consists of states, events, transitions and actions. Each State has a (possibly empty) state-entry and a State exit action that is executed upon State entry or State exit respectively. A transition has a source and a target State and is performed when the State machine is in the source State and the event associated with the transition occurs. For a transition t for event e between State A and State B executing transition t (assuming the FSM is in State A and e occurred) would mean: (1) execute the exit action of State A, (2) execute the action associated with t, (3) execute the entry action of State B and (4) set State B as

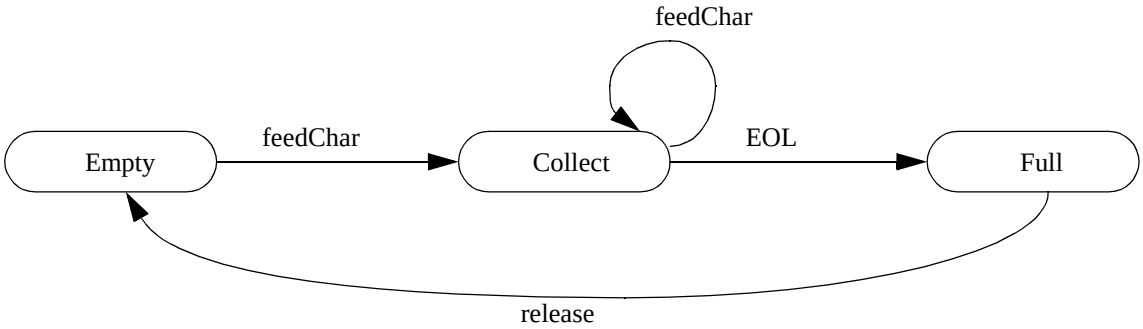

FIGURE 1. WrapAText: a FSM for wrapping text

the current state.

Mostly the State pattern [5] or a variant of this pattern is used to implement FSMs in OO languages like Java and C++. The State pattern has its limitations when it comes to maintenance, though. Also there are two other issues (FSM instantiation and data management) that have to be dealt with. Those issues are further explained in section 3. In this paper we examine these problems and provide a solution that addresses these issues. Also we present a framework that implements this solution and a tool that allows developers to generate a FSM from a specification.

The remainder of this paper is organized as follows: In section 2, two ways of implementing a FSM are introduced: the procedural approach and the State pattern. In section 3 issues with both approaches to implementing FSMs are discussed. In Section 4, a solution is described for these issues and our framework, that implements the solution, is presented. A tool for configuring our framework using XML is presented in section 5. In section 6 assessments are made about our framework. Related work is presented in section 7. We conclude our paper in section 8.

2 Implementing a FSM

As a running example we will use a simple FSM called WrapAText. In figure 1 the diagram for this FSM is presented. The purpose of this FSM is very simple. It inserts a newline in the text after each 80 characters. To do this it has three states representing a line of text. In the Empty State (which also is the default State for this FSM), the FSM waits for characters to be put into the FSM. Once a character is received (feedChar), it moves to the Collect State where it waits for more characters. If 80 characters have been received it moves to the Full State using the EOL (end of line) event. The line is printed on the standard output and the FSM moves back to the Empty State (release event) for the next line of text.

2.1 Procedural Languages

In procedural languages FSMs can be implemented efficiently using a double case statement. The outer case statement selects on the current State of the FSM. Then the inner case statement selects the appropriate behavior for the current State given the type of event that was sent to the FSM (also see the example below). Of course it can also be done the other way around (i.e. first select the event and then the state). What's most efficient depends on the number of events and states and on the likelihood that events or states are added.

```
switch(state)
        case Empty: switch(event)
                case feedChar: ...
                case EOL: ...
                case release: ...
        case Collect: switch(event)
                case feedChar: ...
                case EOL: ...
                case release: ...
        case Full: switch(event)
                case feedChar: ...
                case EOL: ...
                case release: ...
```

Suppose the system is in the Empty State and a feedChar event is received. The outer case statement

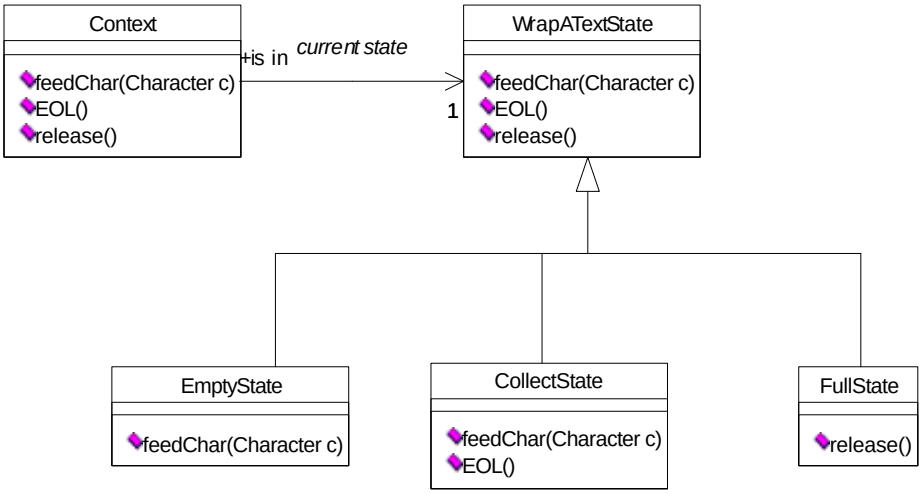

FIGURE 2. The state-pattern.

first selects Empty. Then the inner case statement defined at that point selects feedChar. At that point the behavior for the transition from Empty to Collect is defined. In this approach, a lot of code is duplicated and reuse of code is very difficult. Also redundant code is added (note that the Empty states also has behavior for the EOL and release events even though these events do not occur in this state). The only way to let two states do the same thing for a certain event (this would be useful for the feedChar event) is to copy the code between the two cases. Also the code for entry and exit actions must be duplicated for transitions leaving a certain State or entering a certain state. In addition to this the duplicated code can make maintenance very tedious:

- Bugs have to be fixed more than once
- It is easy to forget a piece of code
- The code becomes very complex.

Of course some tricks can be applied that make the code a bit more modular. One could for instance move the actual implementation to separate procedures. This way, that code no longer has to be copied. The procedure calls, however, still need maintenance. Also naming can be a problem. A name that makes a lot of sense at first might not be suitable after a change. If, for instance, the code for event x in State y is put in a procedure called `state_y_event_x()`, a different name will have to be used if the transition triggered by the event has to be moved to another State (because of a source State change).

The procedural paradigm is an unattractive way to implement a FSM. The main reason it is used anyway is that it gives the best performance and that in some cases the implementations are going to be used over long periods of time without a change. This justifies investing the time and effort to build an implementation in a procedural language. Also code size can be an issue. In embedded machines for instance, memory size and processor capacity are limited.

2.2 The OO Approach

By using object orientation, the use of case-statements can be avoided through the use of dynamic binding. Usually some form of the State pattern is used to model a finite State machine (FSM) [5]. Gamma et al motivate using the State Pattern as follows: *"An object 's behavior depends on its State, and it must change its behavior at run-time depending on that State"* and *"Operations have large, multipart conditional statements that depend on the object's State"* [5]. In other words, each time case statements are used in a procedural language, the State pattern can be used to solve the same problem in an OO language. Each case becomes a State class and the correct case is selected by looking at the current state-object. Each State is represented as a separate class. All those state-classes inherit from a State class. In figure 2 this situation is shown for the WrapAText example. The Context offers an API that has a method for each event in the FSM. Instead of implementing the method

the Context delegates the method to a State class. For each State a subclass of this State class exists. The context also holds references to variables that need to be shared among the different State objects.
At run-time Context objects have a reference to the current State (an instance of a State subclass). In the WrapAText example, the default State is Empty so when the system is started Context will refer to an object of the class EmptyState. The feedChar event is delivered to the State machine by calling a method called feedChar on the context. The context delegates this call to its current State object (EmptyState). The feedChar method in this object implements the State transition from Empty to Collect. When it is executed it changes the current State to CollectState in the Context.

3 Issues in Implementing FSMs

We have studied ways of implementing FSMs in OO languages and identified three issues that we believe should be addressed:

- *Evolution of FSM implementations.* We found that the structure of a FSM tends to change over time and that implementing those changes is difficult using existing FSM implementation methods.
- *FSM instantiation*. Often a FSM is used more than once in a system. To save resources, techniques can be applied to prevent unnecessary duplication of objects.
- *Data management.* Transitions have side effects (actions) that change data in the system. This data has to be available for all the transitions in the FSM. In other words the variables that store the data have to be global. This poses maintenance issues.

In the following sections these issues are discussed in more detail.

3.1 FSM Evolution

Like all software, finite State machine implementations are subject to change. Changes are typically done as maintenance to a system. In this section, we discuss several changes for a FSM and the impact that these changes have on different implementations of a FSM.

- *Adding states*. This usually also means adding/changing transitions.
- *Removing states*. This affects all the transitions to and from the state.
- *Changing states.* Changing a State can mean changing the actions associated with State entry and State exit.
- *Adding events*. This usually happens in combination with other changes such as adding transitions and states
- *Adding transitions.* When adding transitions, it is possible that code should be reused (for instance entry/exit actions)
- *Removing transitions*. This does not affect any states or other transitions.
- *Changing transitions.* Changing a transition can mean changing the source or target state; the event that triggers the transition or the action that takes place when the transition is executed.

These are all typical modifications that happen in the lifecycle of a FSM. Ideally an implementation of a FSM should make it very easy to incorporate these modifications. Unfortunately, this is not the case for the procedural style of implementing nor the State pattern. To illustrate FSM-evolution we changed our running example in the following way:

- We added a new State called Checking
- We changed the transition from Collect to Collect in a transition from Collect to Checking
- We added a transition from Checking to Collect. This also introduced a new event: notFull.
- We changed the transition from Collect to Full in a transition from Checking to Full

The resulting FSM is shown in figure 3. All these changes capture most of the typical maintenance actions that can be performed on FSM implementation.

3.1.1 Procedural Approach

In a system implemented as described in 2.1, the changes mentioned above would be done as follows. First a new case for the Checking State would be added in the outer case statement. For this case, a

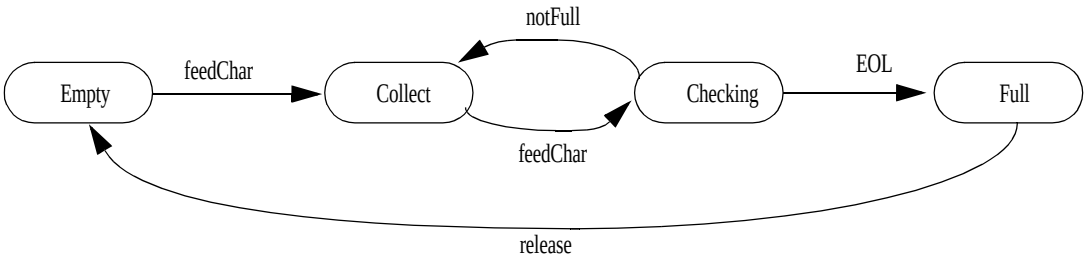

FIGURE 3. The changed WrapAText FSM

new case statement is inserted to select the correct behavior for the events (excluding the new notFull event, we do that later). Then we have to find the feedChar code for the Collect State and change it so that it sets Checking as the new State after the transition. Then we can add a new transition from Checking to Collect. Since this introduces a new event, we will have to change all the inner case statements to support the new event. Finally, the code for the transition from Collect to Full must be changed to set Checking as the new state.
The following aspects of the source code were changed:
- A new case was inserted in the outer case statement.
- A new inner case statement was created for that case.
- All the inner case statements were changed to support a new event.
- The code for the feedChar event in the Collect case was changed.
- The code for the EOL event in the Collect case was changed.

As can be seen in this example, even a simple change such as adding a State to the State machine results in a lot of changes in the code. One can imagine that more complex changes to FSMs that are much larger than our example, are even harder to manage because that would require a lot of changes to the source code.

Problem. A lot of code has to be changed/added to do these simple changes. This makes it hard to do any maintenance activities. Also a lot of code is copied between the different cases. If there is a bug in one of the cases all its copies will have to be fixed too.

Cause. There is no mechanism to share code other than putting behavior in procedures. By using the case statements all events are considered for every State (even the ones that won't occur like EOL in the empty state). This causes a lot of dead code (code that will not be executed). Finally, the case statements also cause the code to be complex.

3.1.2 The State Pattern

The implementation of WrapAText using the State pattern (figure 2) is a lot easier to understand. To do the changes mentioned above the following steps are necessary: First a new subclass of WrapATextState needs to be created for the new State (CheckingState). The new CheckingState class inherits all the event methods from its superclass. Next the CollectState's feedChar method needs to be changed to set the State to CheckingState after it finishes.
To change the source State of the transition between Collect and Full, the contents of the EOL method in CollectState needs to be moved to the EOL method in CheckingState. To create the new transition from Checking to Collect a new method needs to be added to WrapATextState: notFull(). This method can have an empty body in that State (or whatever is applicable to indicate that the method should not be called in that state). The new method is automatically inherited by all subclasses. To let the method perform the transition its behavior will have to be overruled in the CheckingState class. The new method also has to be added to the Context class (making sure it delegates to the current state).
The following things had to be changed in the source code change the FSM:
- A new class was created

- Methods in the CollectState were changed
- A method was added to the WrapATextState superclass
- The State context needs to be changed to support the new event

Problem. Code for a transition can be scattered vertically in the class hierarchy. This makes maintenance of transitions difficult since multiple classes are affected by the changes. Another problem is that methods need to be edited to change the target state. Editing the source State is even more difficult since it requires that methods are moved to another State class. Several classes need to be edited to add an event to the FSM. First of all the Context needs to be edited to support the new event. Second, the State super class needs to be edited to support the new event. Finally, in some State subclasses behavior for transitions triggered by the new event must be added.

Cause. We believe that the main cause for these problems is that the State pattern does not offer first-class representations for all the FSM concepts (also see section 2). Of all FSM concepts, the only concept explicitly represented in the State pattern is the State. The remainder of the concepts are implemented as methods in the State classes (i.e. implicitly). Input events are represented as method headers, output events as method bodies. Entry and exit actions are not represented but can be represented as separate methods in the State class. The responsibility for calling these methods would be in the context where each method that delegates to the current State would also have to call the entry and exit methods. Since this requires some discipline of the developer it will probably not be done correctly.
Since actions are represented as methods in State classes, they are hard to reuse in other states. By putting states in a State class-hierarchy, it is possible to let related states share output events by putting them in a common superclass. But this way, actions are still tied to the State machine. It is very hard to use the actions in a different FSM (with different states). The other FSM concepts (input events, transitions) are represented implicitly. Input events are simulated by letting the FSM context call methods in the current State object. Transitions are executed by letting the involved methods change the current State after they are finished. The disadvantage of not having explicit representations of FSM concepts is that it makes translation between a FSM design and its implementation much more complex. Consequently, when the FSM design changes it is more difficult to synchronize the implementation with the design.

3.2 FSM Instantiation

Sometimes it is necessary to have multiple instances of the same FSM running in a system. In the TCP protocol, for example, up to approximately 30000 connections can exist on one system (one for each port). Each of these connections has to be represented by its own FSM. The structure of the FSM is exactly the same for all those connections. The only unique parts for each FSM instance are the current State of each connection and the value of the variables in the context of the connection's FSM. It would be inefficient to just clone the entire State machine (all the State objects), each time a connection is opened. The number of objects would explode. Suppose the TCP FSM has 25 states, with 30.000 active connections the system would have 30.000 X 25 = 615.000 objects. The memory usage of such a system would be unacceptable.
To contrast the memory usage: a system implemented in a procedural language can implement this as a module with a case statement. This module is loaded into memory only once and all connections share this implementation. The FSM specific data is passed in the form of parameters. Probably some sort of struct can be used to put the FSM specific data in.
Also, a system where the FSM is duplicated does not perform very well because object creation is an expensive operation. In the TCP example, creating a connection requires the creation of approximately 25 objects (states, transitions), each with their own constructor. To solve this problem a mechanism is needed to use FSM's without duplicating all the State objects. The State pattern does not support this directly. This feature can be added, however, by combining the State pattern with the Flyweight pattern [5]. The Flyweight pattern allows objects to be shared between multiple contexts. This prevents that these objects have to be created more than once. To do this, all context specific data has to be removed from the shared objects' classes. We will use the term FSM-instantiation for

the process of creating a context for a FSM. As a consequence, a context can also be called a FSM instance. Multiple instances of a FSM can exist in a system.

3.3 Managing Data in a FSM

Another issue in the implementation of FSMs is data storage. The actions in the transitions of a State machine perform operations on data in the system. These operations change and add variables in the context. If the system has to support FSM instantiation, as described in the previous section, the data has to be separated from the transitions, since this allows each instance to have its own data but share the transitions with the other instances.

3.3.1 Procedural Approach

In the procedural approach this is not so much an issue. Since there are no objects in a procedural language, the data is simply stored in some sort of data structure. Each instance simply consists of a few of those data structures.

3.3.2 State Pattern

In the State pattern, however, this is an issue. The natural place to store data in the State pattern would either be a State class or the context. The disadvantage of storing data in the State objects is that the data is only accessible if the State is also the current state. In other words: after a State change the data becomes inaccessible until the State is set as the current State again. Also this requires that each instance has its own State objects (which prevents effective FSM instantiation). Storing the data in the Context class solves both problems and makes it possible to use the Flyweight pattern for the State classes[1]. Effectively the only class that needs to be instantiated is the Context class. If this solution is used, all data is stored in class variables of the Context class. Storing data in a central place generally is not a good idea in OO programming.

Yet, it is the only way to make sure all transitions in the FSM have access to the same data. So this approach has two disadvantages: It forces the central storage of data and to create a FSM one has to create a subclass of Context (to add all the variables). This makes maintenance hard. In addition, it makes reuse hard, because the methods in State classes are dependent on the Context class and cannot be reused with a different Context class (a different FSM for instance).

3.4 Summary

There are several problems with the procedural way of implementing FSMs:

- Maintenance of FSM implementations is hard because large case/if statements are needed to implement it. This makes the code very complex and often requires editing in multiple places, even for simple changes.
- The only way to reuse code is to copy it. Even if the code is placed in a separate procedure the procedure call still needs to be copied. It also makes the implementation more complex.

These problems can be partly solved by using the State pattern. This however also introduces a number of other problems:

- Though the code is more structured, the use of inheritance causes the code for transitions to be scattered throughout the State class hierarchy. This makes maintenance of those transitions hard.
- Changing target/source states of a transition requires changing or even moving methods to other classes.
- To add events to the FSM several classes need to be edited.
- There is no convenient way to let transitions share data.

4 Novel Approach to FSM Implementation

Several causes can be found for the problems with the State pattern mentioned in section 3:

1. The Flyweight pattern [5] can be used to share objects. It is useful if it is unfeasible to instantiate a class each time a object of that class is needed. The Flyweight pattern is especially applicable if the class does not have any data that needs to be shared.

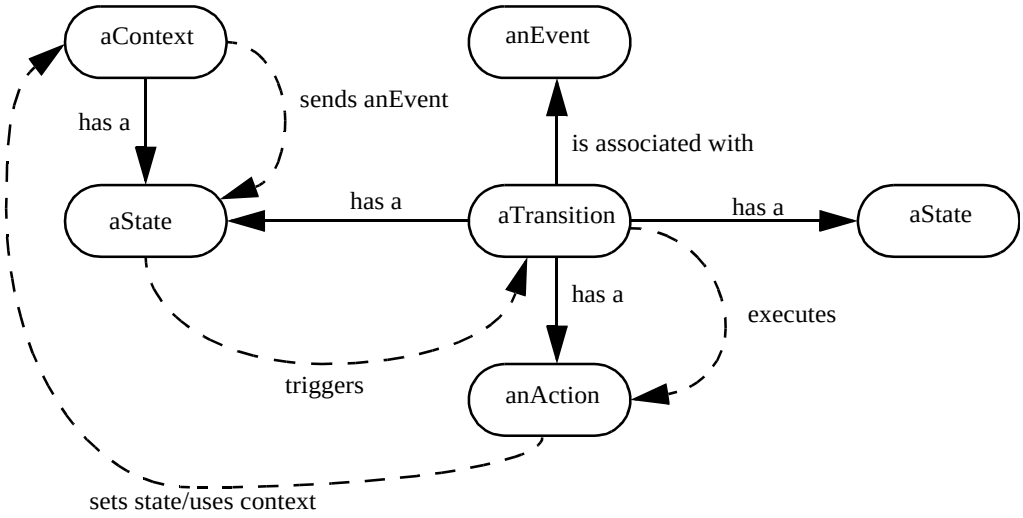

FIGURE 4. The FSM Framework's components.

- The State pattern does not provide explicit representations for all the FSM concepts. This makes maintenance hard because it is not obvious how to translate a design change in the FSM to the implementation and a design-change may result in multiple implementation elements being edited.
- The State pattern is not blackbox. Building a protocol requires developers to extend classes rather than to configure them. To do so, code needs to be edited and extended rather than composed from existing components.
- Implementations are complex because the translation from design to implementation causes transitions and events to be scattered over multiple classes and methods.
- The State pattern prevents reuse of behavior by integrating FSM concepts in a single class. The State classes in the State pattern host a state, transitions from that State and events (in the form of method signatures). This makes reuse of either of these concepts independent of the others hard because they are tied together. The inheritance hierarchy for the State classes complicates things further because transitions (and events) can be scattered throughout the hierarchy.

Most of these causes seem to point at the lack of structure in the State pattern (structure that exists at the design level). This lack of structures causes developers to put things together in one method or class that should rather be implemented separately. The solution we will present in this section will address the problems by providing more structure at the implementation level.

4.1 Conceptual Design

To address the issues mentioned in above we modeled the FSM concepts as objects. We have created a design that also deals with FSM evolution and instantiation. The implication of this is that most of the objects in the design must be sharable between FSM instances. This also implies that those objects cannot store any context specific data. An additional goal for the framework was to allow blackbox configuration[2]. The rationale behind this was that it should be possible to separate a FSM's structure from its behavior (i.e. transition actions or State entry/exit actions). In figure 4 the conceptual model of our FSM framework is presented. The rounded boxes represent the different components in the framework. The solid arrows indicate association relations between the components and the dashed arrows indicate how the components use each other.

Similar to the State pattern, there is a Context component that has a reference to the current state. The latter is represented as a State object (rather a State subclass in the State pattern). The key concept in the framework is a transition. The transition object has a reference to the target State and an Action object. For the latter, the Command pattern [5] is used. This makes it possible to reuse commands in multiple places in the framework. A State is associated with a set of transitions. The FSM responds to

2. Blackbox frameworks provide components in addition to the white box framework (abstract classes + interfaces). Components provide a convenient way to use the framework. Relations between blackbox components can be established dynamically.

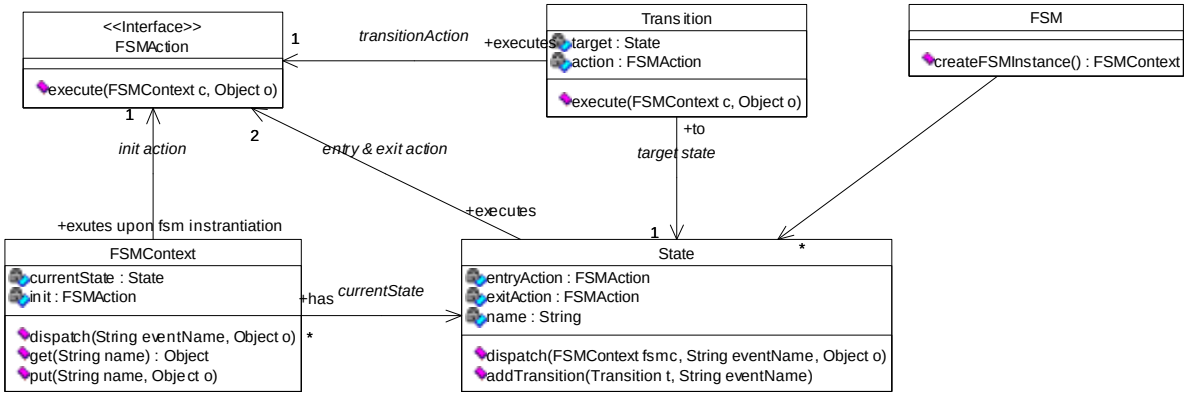

FIGURE 5. Class diagram for the FSM Framework

events that are sent to the context. The context passes the events on to the current state. The State maintains a list of transition, event pairs. When an event is received the corresponding transition is located and then executed (triggered). The transition object simply executes the associated action and then sets the target State as the current State in the context.

To enable FSM instantiation in an efficient way, no other objects than the context may be duplicated. All the State objects, event objects, transition objects and action objects are created only once. The implication of this is that none of those objects can store any context specific data (because they are shared among multiple contexts). When, however, an action object is executed (usually as the result of a transition being triggered), context specific data may be needed. The only object that can provide access to this data is the context. Since all events are dispatched to the current State by the context, a reference to the context can be passed along. The State in its turn, passes the reference to the transition that is triggered. The transition finally gives the reference to the action object. This way the action object can have access to context specific data without being context specific itself.

A special mechanism is used to store and retrieve data from the context. Normally, the context class would have to be sub-classed to contain the variables needed by the actions in the FSM. This effectively ties those actions to the context class, which prevents reuse of those actions in other FSMs since this makes the context subclasses FSM specific. To resolve this issue we turned the context into a object repository. Actions can put and get variables in the context. Actions can share variables by referring to them under the same name. This way the variables do not have to be part of the context class. Initialization of the variables can be handled by a special action object (init) that is executed when a new context object is created. Action objects can also be used to model State entry and exit actions.

4.2 An Implementation

We have implemented the design described in the previous section as a framework [8] in Java. We have used the framework to implement the WrapAText example and to perform performance assessments (also see section 6). The core framework consists of only four classes and one interface. In figure 5, a class diagram is shown for the framework's core classes. We'll shortly describe the classes here:

- *State*. This class models the states in the FSM. Each State has a name that can be set as a property in this class. State also provides a method to associate events with transitions. It also provides a dispatch method to trigger transitions for incoming events. The dispatch method (after finding the right transition) first executes the state-exit action. Subsequently the transition is executed and then the target State in the transition is set as the current State in the context. Finally the state-entry method for the target State is executed.
- *FSMContext*. This class maintains a reference to the current State (is generally changed by executing transitions) and functions as an object repository for actions. Actions can store objects in the context using the put method. The objects can later be retrieved using the get method. Whenever a

```
        <states>
                <State name="Empty"/>
                <State name="Collect" initaction="collectEntry.ser"/>
                <State name="Full" initaction="fullEntry.ser"/>
        </states>
        <events>
                <event name="feedChar"/>
                <event name="EOL"/>
                <event name="release"/>
        </events>
        <transitions>
                <transition sourcestate="Empty"
                            targetstate="Collect"
                            event="feedChar"
                            action="processChar.ser"/>
                <transition sourcestate="Collect"
                            targetstate="Collect"
                            event="feedChar"
                            action="processChar.ser"/>
                <transition sourcestate="Collect"
                            targetstate="Full"
                            event="EOL"
                            action="skip.ser"/>
                <transition sourcestate="Full"
                            targetstate="Empty"
                            event="release"
                            action="reset.ser"/>
        </transitions>
</fsm>
```

FIGURE 6. WrapAText specified in XML

new FSMContext object is created (FSM instantiation), the init action is executed. This action can be used to pre-define variables for the actions in the FSM.

- *Transition*. The transition object has only one method: execute(). This method is called by a State when an event is dispatched that triggers the transition.
- *FSM*. This class functions as a central point of access to the FSM. It provides methods to add states, events and transitions. It also provides a method to instantiate the FSM (resulting in the creation and initialization of a new FSMContext object).
- *FSMAction*. This interface has to be implemented by all actions in the FSM. It functions as an implementation of the Command pattern as described in [5].

5 Our FSM Framework Configuration Tool

In [12] a typical evolution path of frameworks is described. According to this paper, frameworks start as white box frameworks (just abstract classes and interfaces). Gradually components are added and the framework evolves into a black box framework. One of the later steps in this evolution path is the creation of configuration tools. Our FSM Framework consists of components thus creating the possibility of making such a configuration tool. A tool significantly eases the use of our framework. since developers only have to work with the tool instead of complex source code. As a proof of concept, we have built a tool that takes a FSM specification in the form of an XML document [17] as an input. XML is suitable for modeling hierarchical data, such as a FSM. The wide support for this language makes it easier to built graphical tools that work on the XML specification of a FSM.

5.1 FSMs in XML

In figure 6 an example of an XML file is given that can be used to create a FSM. In this file the WrapAText FSM in figure 1 is specified. Several tags like `<states>` and `<transition>` are used. The structure of the file is very straightforward. We did not create a DTD[3] for FSMs but that would be necessary if the tool would be used in a wider context.

3. Document Type Definition. DTDs can be used to validate XML document (that is check whether the structure of the document is valid against the rules specified in the DTD).

A problem in specifying FSMs using XML is that FSMActions cannot be modeled this way. The FSMAction interface is the only whitebox element in our framework and as such is not suitable for configuration by a tool. To resolve this issue we developed a mechanism where FSMAction components are instantiated, configured and saved to a file using serialization. The saved files can be referred to from the XML file. When the framework is configured the FSMAction component files are deserialized by the configuration tool and plugged into the FSM framework. Alternatively, we could have used the dynamic class-loading feature of Java. This would, however, prevent the configuration of any parameters the actions may contain. The dynamic abilities of Java are essential in this tool since a static language such as C++ does not have a mechanism for serialization (though it is possible to simulate it to some extent), nor does it have a mechanism for loading classes dynamically.

5.2 Configuring and Instantiating FSMs

The FSMGenerator, as our tool is called, uses the IBM xml4j parser [18]. It parses a document like the example in figure 6. After the document is parsed, the parse tree can be accessed using the Document Object Model API that is standardized by the World Wide Web Consortium (W3C) [19]. Using this interface, the tool traverses the object hierarchy and configures the components in our framework. The generator first retrieves all the State tags and uses the FSM class to create a State for each of them. Then it proceeds with the event tags and after that the transition tags. After it is finished the tool gives back a FSM object that contains the FSM as specified in the XML document. The parser also has the ability to process DTD's so if a DTD was developed for XML code that specifies a FSM, the parser would automatically check whether an input file is correct. The DOM API can also be used to create XML. This feature would be useful if a graphical tool were developed. Through the DOM API, such an application can create XML files. The FSM object can be used to create FSM instances.

5.3 WrapAText in the FSM Framework

Describing the FSM in XML is pretty straightforward, as can be seen in figure 6. Most of the implementation effort is required for implementing the FSMAction objects. Once that is done, the FSM can be generated (at run-time) and used. Five serialized FSMAction objects are pre-defined. Since the FSM framework allows the use of entry and exit actions in states, they are used those where appropiate. The processChar action is used in two transitions. This is where most of the work is done. To illustrate the implementation of a typical FSMAction, the source code for one of the actions is shown here:

```
public class ProcessChar implements FSMAction, java.io.Serializable {
    public ProcessChar() {
    }

    public void execute(FSMContext fsmc, Object o) {
       Counter c = (Counter)fsmc.get("counter");
       c.increment();
       StringBuffer line = (StringBuffer)fsmc.get("line");
       line.append("" + o);
    }
}
```

The FSMAction uses the FSMContext to retrieve two variables (a counter and the line of text that is presently created) that are retrieved from the context. Also the Serializable interface is implemented to indicate that this class can be serialized.

6 Assessment

In section 3, we evaluated the implementation of finite State machines using the procedural approach and the State pattern. This evaluation revealed a number of problems, based on which we developed an alternative approach. In this section we evaluate our approach with respect to maintenance and performance.

6.1 Maintenance

The same changes we applied in section 3.1 can be applied to the implementation of WrapAText in the FSM framework. We'll use the implementation as described in section 5.3 to apply the changes

to. All of the changes are restricted to editing the XML document since the behavior as defined in the FSMActions remains more or less the same.
To add the Checking state, we add a line to the XML file:

```
< State name="Checking" />
```

Then we change the target State of the Collect to Collect transition by changing the definition in the XML file. We do the same for the Collect to Full transition. The new lines look like this:

```
< transition sourcestate="Collect" targetstate="Checking" event="feedChar"
  action="processChar.ser" />
< transition sourcestate="Checking" targetstate="Full" event="EOL" action="skip.ser" />
```

Then we add the transition from Checking to Collect:

```
<transition sourcestate="Checking" targetstate="Collect" event="notFull" action="skip.ser" />
```

Finally the entry action of Collect is moved to the Checking State by setting the initaction property in Checking and removing that property in Collect. Changing a FSM implemented in this style does not require any source editing (except for the XML file of course) unless new/different behavior is needed. In that case the changes are restricted to creating/editing FSMActions.

6.2 Performance

To compare the performance of the new approach in implementing FSMs to a traditional approach using the State pattern, we performed a test[4]. The performance measurements showed that the FSM Framework was almost as fast as the State pattern for larger State machines but there is some overhead. The more computation is performed in the actions on the transitions that are executed, the smaller the performance gap.
To do the performance measurements, the WrapAText FSM implementation was used. This is a very easy case to implement since most of the actions are quite trivial. Some global data has to be maintained: a String to collect received characters and a counter to count the characters. Two implementations of this FSM were created: one using the State Pattern and one using our FSM Framework presented earlier.
Several different measurements were performed. First, we measured the FSM as it was implemented. This measurement showed that the program spent most of its time switching State since the actions on the transitions are rather trivial. To make the situation more realistic loops were inserted into the transition actions to make sure the computation in the transitions actually took some time (more realistic) and the measurements were performed again.
Four different measurements (see figure 7) were done:

- Measuring how long it takes to process 10,000,000 characters (I)
- The same as (I) but now with a 100 cycle for loop inserted in the feedChar code. Each time a character is processed, the loop is executed (II)
- The same as (II) with a 1000 cycle loop (III)
- The same as (II) with a 10000 cycle loop (IV)

The loop ensures that processing a character would take some time. This simulates a real world situation where a transition takes some time to execute.
In figure 7, a diagram our measurements is shown. Each case was tested for both the State pattern and the FSM framework. For each test, the time to process the characters was measured (in seconds). The bars in the graph illustrate the relative performance difference. Not surprisingly the performance gap decreases if the amount of time spent in the actions on a transition increases. The numbers show that a State transition in the FSM Framework (exclusive action) is about twice as expensive as in the State Pattern implementation for simple transitions. The situation becomes better if the transitions become more complex (and less trivial). The reason for this is that the more complex the transitions are the smaller the relative overhead of changing State is. This is illustrated by case IV where the performance difference is only 13%.
In general one could say that the State pattern is more efficient if a lot of small transitions take place in a FSM. The performance difference becomes negligible if the actions on the transitions become

4. We did not test the procedural solution because we expected it to be faster than both the State pattern and the FSM framework, whereas its maintainability problems virtually prohibit its use in evolving systems..

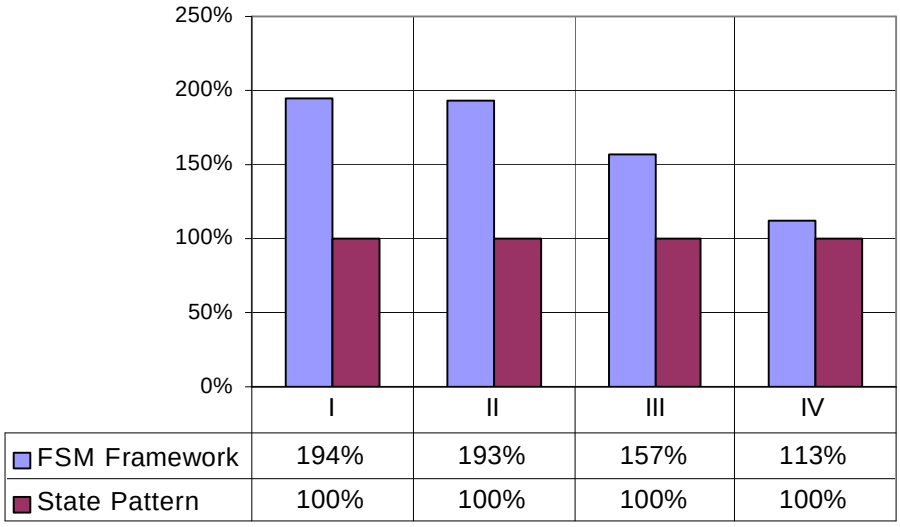

FIGURE 7. Performance measurements (measurements in seconds)

more computational intensive. Consequently, for larger systems, the performance difference is negligible. Moreover since this is only a toy framework, the performance gap could be decreased further by optimizing the implementation of our framework. The main reason why State transitions take longer to execute is that the transition object has to be looked up in a hashtable object each time it is executed. The hashtable object maps event names to transitions.

7 Related Work

State Machines in General. FSM have been used as a way to model object-oriented systems. Important work in this context is that of Harel's Statecharts [7] and ObjChart [6]. ObjChart is a visual formalism for modeling a system of concurrently interacting objects and the relations between these objects. The FSMs that this formalism delivers are too fine-grained (single classes are modeled as a FSM) to implement using our FSM Framework. Rather our framework should be used for more coarse-grained systems where the complex structure is captured by a FSM and the details of the behavior of this machine are implemented as action objects. Most of these approaches seem to focus on modeling individual objects as FSMs rather than larger systems.

FSM Implementation. In the GoF book [5] the State pattern is introduced. In [4], Dyson and Anderson elaborate on this pattern. One of the things they add is a pattern that helps to reduce the number of objects in situations where a FSM is instantiated more than once (essentially by applying the fly-weight pattern). In [11], a complex variant of the State Pattern called MOODS is introduced. In this variant, the State class hierarchy uses multiple inheritance to model nested states as in Harel's Statecharts [7]. In [10], the State pattern is used to model the behavior of reactive components in an event centered architecture. Interestingly it is suggested that a event dispatcher class for the State machine can be generated automatically.

In [13] an implementation technique is presented to reuse behavior in State machines through inheritance of other State machines. The authors also present an implementation model that is in some ways similar to the model presented in this paper. Our approach differs from theirs in that it factors out behavior (in the form of actions). The remaining FSM is more flexible (it can be changed on the fly if needed). Our approach establishes reuse using a high level specification language for the State machine and by using action components, that are in principle independent of the FSM. Bosch [3] uses a different approach to mix FSMs with the object-orientation paradigm. Rather than translating a FSM to a OO implementation a extended OO language that incorporates states as first class entities is used.

Yet another way of implementing FSMs in an object-oriented way is presented in [1]. The implementation modeled there resembles the State pattern but is a slightly more explicit in defining events and transitions. It still suffers from the problem caused by actions being integrated with the State classes. Also data management and FSM instantiation are not dealt with. The author also recognizes the need

for a mapping between design (a FSM) and implementation like there is for class diagrams. This need is also recognized in [2], where several issues in implementing FSMs are discussed.

Event Dispatching. Event dispatching is rudimentary in the current version of our framework at this moment. A better approach can be found [14], where the Reactor pattern is introduced. An important advantage of the way events are modeled in our framework, however, is that they are blackbox components. The Reactor pattern would require one to make subclasses of some State class. A different approach would be to provide a number of default events as presented in [9], where the author classifies events in different groups.

Frameworks. A great introduction to frameworks can be found in [8]. In this thesis several issues surrounding object-oriented frameworks are discussed. A pattern language for developing frameworks can be found in [12]. One of the patterns that is discussed in this paper is the Black box Framework pattern which we used while creating our framework. Another pattern in this article is called Language Tools applies to our configuration tool.

8 Conclusion

The existing State pattern does not provide explicit representations for all the FSM concepts. Programs that use it are complex and it cannot be used in a blackbox way. This makes maintenance hard because it is not obvious how to apply a design change to the implementation. Also support for FSM instantiation and data management is not present by default. Our solution however, provides abstractions for all of the FSM concepts. There is a State class, an Event class (input events), a Transition class, a FSMAction interface (output events). Furthermore, each State has state-entry and state-exit actions. In addition to that it supports FSM instantiation and provides a solution for data management that allows to decouple behavior from the FSM structure.The latter leads to cross FSM, reusable behavior.

The State pattern is not blackbox and requires source code to be written in order to apply it. Building a FSM requires the developer to extend classes rather than to configure them. Alternatively, our FSM Framework[5] can be configured (with a tool if needed) in a blackbox way. Only FSMActions need to be implemented in our framework. The resulting FSMAction objects can be reused in other FSMs. This opens the possibility to make a FSMAction component library.

Our approach has several advantages over implementing FSMs using the State pattern, including:

- States are no longer created by inheritance but by configuration. The same is the case for events. Also, the context can be represented by a single component.
- Inheritance is only applied where it is useful: extending behavior. Related actions can share behavior through inheritance. Also actions can delegate to other actions (removing the need for events supporting more than one action).
- States, actions, events and transitions now have explicit representations. This makes the mapping between a FSM design and implementation more direct and consequently easier to use. A tool could create all the event, State and context objects by simply configuring them. All that would be required from the user would be implementing the actions.
- It is possible to configure FSMs in a blackbox way. This can be automated by using a tool such as our FSMGenerator.

There are also some disadvantages compared to the original State pattern:

- The context repository object possibly causes a performance penalty compared to directly accessing variables, since variables need to be obtained from a repository. However a pretty efficient hashtable implementation is used. The measurements we performed showed that the performance gap with the State pattern decreases as the transitions become more complicated.
- It could be difficult to keep track of what's going on in the context. The context is simply a large repository of objects. All actions in the FSM read and write to those objects (and possibly add new ones). This can, however, be solved by providing tracing and debugging tools.

5. To receive a copy of our framework, documentation and some examples, contact the first author.

8.1 Future work

Our FSM framework can be extended in many ways. An obvious extension is to add conditional transitions. Conditional transitions are used to specify transitions that only occur if the trigger event occurs and the condition holds true. Though this clearly is a powerful concept, it is hard to implement it in a OO way. A possibility could be to use the Command pattern again to create condition objects with a boolean method but that would tie the conditions to the implementation thus they can't be specified at the XML level. To solve this problem a large number of standard conditions could be provided (in the form of components).

A next step is to extend our FSM framework to support Statechart-like FSMs. Statecharts are normal FSMs + nesting + orthogonality + broadcasting events [7]. These extensions would allow developers to specify Statecharts in our configuration tool, which then maps the statecharts to regular FSMs automatically. The extensions require a more complex dispatching algoritm for events. Such an extension could be used to make the State diagrams in OO modeling methods such as UML and OMT executable.

Though performance is already quite acceptable, much of our implementation of the framework can be optimized. The bottlenecks seem to be the event dispatching mechanism and the variable lookup in the context. Our current implementation uses hashtables to implement these. By replacing the hashtable solution with a faster implementation, a significant performance increase is likely.

9 References

[1] M. Ackroyd, *"Object-oriented design of a finite State machine"*, Journal of Object Oriented Programming, June 1995.

[2] F. Barbier, Henri Briand, B. Dano, S. Rideau, *"The executability of Object Oriented Finite State Machines"*, Journal of Object Oriented Programming, July/August 1998.

[3] J. Bosch, *"Abstracting Object State"*, Object Oriented Systems, June 1995.

[4] P. Dyson, B. Anderson, *"State Patterns"*, Pattern Languages of Programming Design 3, edited by Martin/Riehle/ Buschmann Addison Wesley 1998

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *"Design Patterns - Elements of Reusable Object Oriented software"*, Addison Wesley, 1995.

[6] D. Gangopadhyay, Subrata Mitra, *"ObjChart: Tangible Specification of Reactive Object Bahvior"*, Proceedings of ECOOP '93, p432-457 July 1993.

[7] D. Harel, *"Statecharts: a Visual Approach to Complex Systems(revised)"*, report CS86-02 Dep. App Math's Weizman Inst. Science Rehovot Israel, March 1986.

[8] M. Mattson, *"Object-Oriented Frameworks – A Survey of Methodological Issues"*, Department of computer science, Lund University, 1996.

[9] J. J. Odell, *"Events and their specification"*, Journal of Object Oriented Programming, July/August 1994.

[10] A. Ran, "Patterms of Events", Pattern Languages of Program Design, edited by Coplien/Schmidt. Addison Wesley, 1995

[11] A. Ran, *"MOODS: Models for Object-Oriented Design of State"*, Pattern Languages of Program Design 2, edited by Vlissides/Coplien/Kerth. Addison Wesley, 1996

[12] D. Roberts, R Johnson, *"Patterns for evolving frameworks"*, Pattern Languages of Program Design 3 (p471-p486), Addison-Wesley, 1998.

[13] A. Sane, R. Campbell, *"Object Oriented State Machines: Subclassing Composition, Delegation and Genericity"*, Proceedings of OOPSLA '95 p17-32, 1995.

[14] D. C. Schmidt, *"Reactor: An Object Behavior Pattern for Concurrent Event Dmultiplexing and Event Handler Dispatching"*, in Coplien, Schmidt, "Pattern Languages of Program Design", Addison Wesley 1995, p529-546.

[15] J. M. Zweig, R. E. Johnson, *"The Conduit: a Communication Abstraction in C++"*, Usenx C++ Conference 1990.

[16] *"Transmission Control Protocol – DARPA Internet Program Protocol Specification"*, RFC 793, September 1981.

[17] http://www.w3c.org/XML/index.html

[18] http://www.alphaworks.ibm.com/Home/index.html

[19] http://www.w3c.org/index.html