

A REPORT ON SMC-THE STATE MACHINE COMPILER

By

Miloud Eloumri

Get state machine classes at a glance

Queen's University

Kingston, Ontario, Canada

(May, 2008)

Copyright ©Miloud Eloumri, 2008

Abstract

State Machine Compiler (SMC) takes as an input a Finite State Machine (FSM) described in a textual file, <fileName.sm> and generates a state pattern classes implementing that FSM in one of the fourteen supported programming languages, as May, 2008, then it integrates the FSM with a specified application class. The supported target languages are:

C	C++	C#	[incr Tcl]
Groovy	Java	Lua	Objective-C
Perl	PHP	Python	Ruby
Scala	VB.net		

SMC is a Java based tool and one of its aims is to increasingly target more programming languages at the same time keeping its syntax and use light and simple. The SMC syntax is the same with all supported languages. SMC also defines several transitions types, including but not limited Default transitions, push/pup transitions, as well as, transition arguments, guards, and actions. Likewise, SMC has a Default state and state's Entry/Exit actions. SMC is not a "real programming language compiler"; instead, it reads its file verbatim and according to the specified target language, it generates the code following the powerful pattern of state pattern.

Acknowledgements

First, I would like to express my great thanks and gratitude to Charles W. Rapp, the creator and owner of SMC SourceForge project, for his outstanding work and sharing SMC as an open software, and also for his immediate assistance and replay to all questions raised on the SMC's forums. Second, I would also like to thank every individual and reference cited in this report. Last but not least, great thanks go to my supervisor Juergen Dingel for his endless help and advice, as well as, for introducing me to this research area.

(Miloud Eloumri)

(May, 2008)

Preface

Specifically, this report tries to give some information regarding State Machine Compiler (SMC). Likewise, the report, generally, discusses some basic aspects of finite state machines (FSMs), as well as their implementation concepts. The report provides simple explanation enhanced with some straightforward examples. It should be mentioned that this report discusses SMC aspects taking in the consideration, primarily Java as a target language. However, SMC is intended to support multiple programming languages with minor changes. The SMC's syntax is the same regardless of the specified target language. In May 20, 2008, while working on this report, a new version of SMC has been released (SMCv.5.1.0). The major updates in this release are: adding support to PHP and Scala programming languages, adding a new transition type: jump transition, fixing some errors. This release is built on Java 5. The next future planned version (SMCv.6.0.0) will be built on Java 6 and will include two distinguish loopback transitions: internal and external. The SMC version that is used in this report is SMCv.5.0.2, released in January 14, 2008. SMC main website, <http://smc.sourceforge.net> provides more rich details regarding SMC and its implementation for all supported programming languages.

Table of Contents

Abstract	ii
Acknowledgements	iii
Preface.....	iv
Table of Contents.....	v
List of Figures.....	vii
List of Tables	viii
Chapter 1 Introduction	9
1.1 Introducing SMC Concepts	9
Chapter 2 Finite State Machines and Diagrams.....	12
2.1 Basics.....	12
2.2 Implementing Finite State Machines	15
2.2.1 Implementing Finite State Machines Using Nested Switch Case Statements or IF Then Else Statements	15
2.2.2 Implementing Finite State Machines Using State Pattern.....	17
2.3 SMC Motivation	19
Chapter 3 SMC: Use and Syntax	21
3.1 SMC in Nutshell.....	21
3.2 SMC Requirements	25
3.3 SMC Compiling	26
3.4 States and Transitions	29
3.4.1 Simple Transition	29
3.4.2 Jump Transition.....	30
3.4.3 Loopback Transition.....	30
3.4.4 Push/Pop Transition.....	31
3.4.5 Transition Actions	38
3.4.6 Transition Arguments	39
3.4.7 Transition Guards	40
3.4.8 Default State and Transition.....	43
3.4.9 Default State.....	44
3.4.10 Default Transition.....	45
3.4.11 State Entry and Exit Actions	47
3.4.12 Transitions Issue Transitions.....	49

Chapter 4 SMC Pattern and Generation.....	51
4.1 SMC Pattern.....	51
4.2 SMC Generated Code.....	53
4.3 Generating GraphViz Dot File.....	66
4.4 Generating HTML Table.....	69
4.5 Persistence.....	70
4.6 State Change Notification.....	71
4.7 Reflection.....	71
Chapter 5 Conclusion.....	72
5.1 Closing Points.....	72
Appendix A SMC EBNF Grammar.....	75

List of Figures

Figure 1: SMC process.....	9
Figure 2: Interaction between the context class and the application class	11
Figure 3: Simple turnstile FSM diagram.....	13
Figure 4: State pattern architecture	17
Figure 5: Turnstile FSM in state pattern	18
Figure 6: (Figure 3 Repeated) Simple turnstile FSM diagram	21
Figure 7: Simple transition.....	29
Figure 8: Loopback transition.....	30
Figure 9: Turnstile FSM with violation state and diagnostic mode	32
Figure 10: Simple transition action argument	39
Figure 11: Transition with arguments, guards, and actions with transition arguments.....	42
Figure 12: Default state.....	44
Figure 13: Turnstile State diagram with Entry and Exit actions.....	48
Figure 14: SMC pattern.....	51
Figure 15: Class diagram for the generated code in Table 16	61
Figure 16: Dependency between generated classes, statemap library and application class, (corresponds to the generated code in Table 16)	63
Figure 17: Application class, .sm file, SMC compiler, generated code (Table 16) and SMC library (statemap) dependency and association	65
Figure 18: Turnstile diagrams with least details (-glevel 0).....	66
Figure 19: Turnstile diagrams with -glevel 1	67
Figure 20: Turnstile diagrams with -glevel 2	67
Figure 21: Turnstile diagram with multiple maps.....	68

List of Tables

Table 1: State transition table	14
Table 2: Nested switch case statements implementing turnstile FSM	16
Table 3: Simple SMC state syntax	22
Table 4: SMC file syntax with application class	23
Table 5: SMC file syntax with explanation comments	24
Table 6: SMC code for simple transitions in Figure 7	29
Table 7: Loopback Transition.....	31
Table 8: Multiple maps with push and pop transitions	35
Table 9: SMC simple transition action.....	39
Table 10: SMC equivalent code of Figure 11	42
Table 11: Defining Default state in SMC language	45
Table 12: Defining Default transitions and Default state in SMC file.....	46
Table 13: Entry and Exit actions execution and transitions types dependency	48
Table 14:Entry and Exit actions in SMC.....	49
Table 16: Generated code: TurnstileAppClassContext.java.....	60
Table 17: Generated HTML table for SMC file in Table 8	69

Chapter 1

Introduction

1.1 Introducing SMC Concepts

The State Machine Compiler, SMC [1] is an open source Java project hosted on sourceforge.net (<http://smc.sourceforge.net>). SMC mates finite state machines (FSMs) with objects. It generates state pattern code representing FSM for individual objects, such that it allows defining a state machine in one textual file which must have (.sm) ending and it represents the states an object can be in, as well as transitions and actions. Figure 1 demonstrates graphically how SMC performs [1].

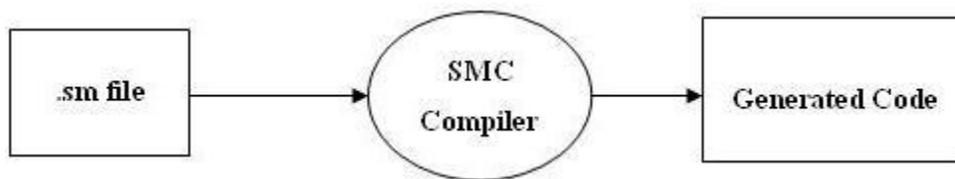


Figure 1: SMC process

SMC's .sm file that should be written is basically a state transition table [1], STT (see state machines and diagrams basic concepts section for more details on STT). In other words, the state machine logic is grouped inside one or more table or block named "map" in which each state is defined along with its transitions, next state, and actions. The .sm file must have only one map that identifies the starting point of the state machine. That is to say, only one map declares the start or initial state of a FSM, but the .sm file may have other maps without having the start state. From this descriptive .sm file, SMC generates state pattern classes in one file. Simply stated, the generated file contains among others the context class and a class for each state, (the generated code will be discussed later in the SMC generated code section). Likewise, the .sm file must

declare a class that will be integrated with the generated classes, namely the generated context class. The declared class which is referred to as “<AppClass> – application class” is the class that a programmer writes in order to interact with the generated context class (or FSM). The generated context class will take the same name as the <AppClass> class name specified within the .sm file but appended with the word “Context”. As well, the generated file that contains all generated classes will have a name that is derived from the .sm file name. Therefore, the names of the .sm file and the <AppClass> class specified in it are required to be the same. This applies, specifically when generating Java code. The reason for this is that Java compiler requires a certain class name to be defined in a file that has the same name as the class name. Since SMC is a Java based tool, it follows Java in the naming principles (see The SMC syntax and use section for further details). Also, the <AppClass> class should define action methods whose names correspond to the actions names in the FSM. These methods should be accessible to the generated context class; hence, the methods should be declared as public or package when generating code for Java if in the same package, as well as, the methods should have a void return type, but if there is a return value, SMC ignores it. The interacting mechanism of the FSMs; specifically the generated context class, to the application class, the <AppClass> is very simple as follows:

- Define and instantiate an instance of the generated context class in the application class.
- Issue transitions by invoking appropriate transition method on the context class.

Hence as mentioned above; on one hand, the generated context class defines transition methods according to the defined transitions in the .sm file. On the other hand, the application class <AppClass> defines action methods whose names must correspond to the actions names in the .sm file. Figure 2 [1] shows the interaction between the generated context class and the application class.

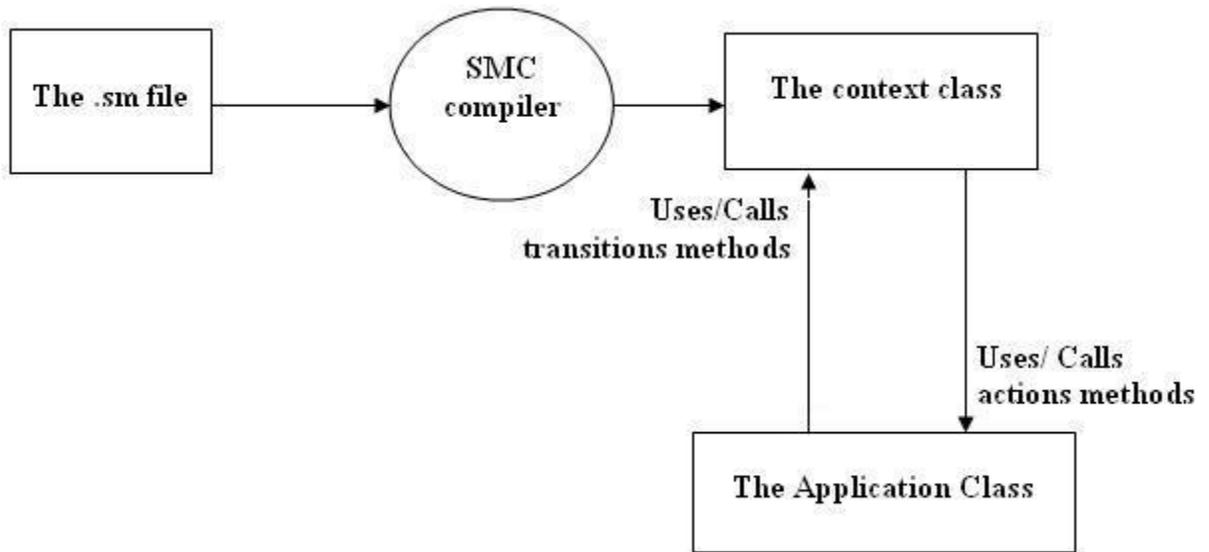


Figure 2: Interaction between the context class and the application class

The generated context class calls action methods on the application class, in turn; the application class invokes transition methods on the context class.

Further, SMC [1] handles unexpected events such that it defines default state and transitions. In addition to the simple and default transitions, SMC also defines a jump, loopback and push/pop transitions. As a convention of state machines, SMC's transitions may have arguments, guards and actions. And, states may contain Entry and Exit actions. These concepts will be introduced subsequently in the following sections using some examples.

Chapter 2

Finite State Machines and Diagrams

2.1 Basics

State machines can be seen in several real world objects [2]; for example, vending machines, washing machines, digital watches and many other electronic devices. State machines simplify implementation of programs that handles input events and states. Specifically, state machines are most useful when developing GUI applications and communication protocols; for instance. State machines when appropriately applied, result in more reliable, less coupled and easy maintained code. So, what is a state machine? Simply stated, “*a state machine is a system with a set of unique states*” [2]. In state machines there are special states; that is, one state is called initial state from which a state machine initializes and one or more of other states are called final states where a state machine exits. Moving from a state to another is done by transitions; thus, transitions connect system states. Each transition represents an input event. The event triggers the transition when it occurs and causes to move from the current state to the new next state with some exceptions such remaining in the same state, as will be illustrated subsequently in the next sections. State machines are often drawn as diagrams [2]. Consider the following Figure 3 [3] that shows a simple state diagram describing FSM for a subway turnstile. This example will be used to introduce the implementation of state machines, as well as SMC aspects. It is helpful to state that Figures and examples in the report are colored. That is, Black is for user defined states and maps names, Red is for transitions names, Purple is for actions names and Blue is for SMC keywords and symbols. It is also helpful to discuss some basic concepts of state machines,

diagrams and their implementation techniques regarding to SMC related concepts, as it is presented in the next section. Simple turnstile FSM diagram

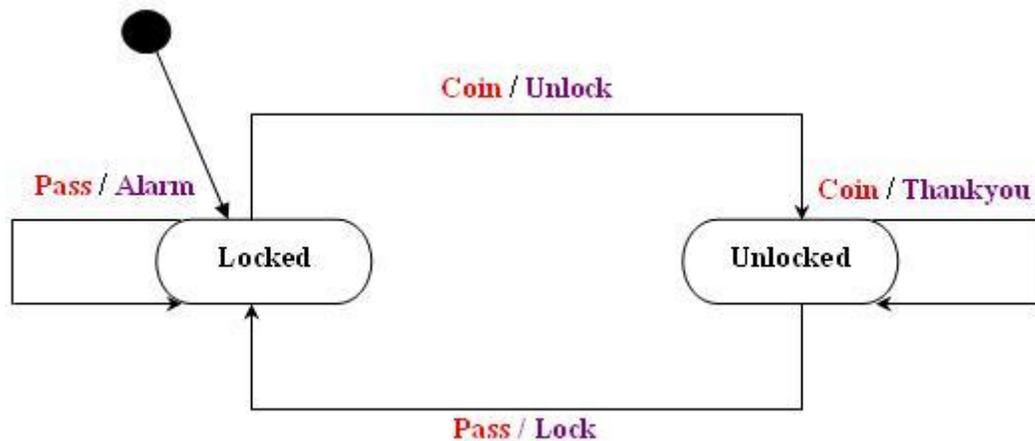


Figure 3: Simple turnstile FSM diagram

In state diagrams, the rounded rectangles represent states [3]. There are two states in Figure 3, **Locked** and **Unlocked**. The black circle that points to the **Locked** state is called the initial pseudo state. It indicates that the **Locked** state is the actual start state for this FSM. Transitions are represented by arrows between states. Each transition has a label of two parts separated by a slash. The first part represents the event name that triggers the transition. The second part represents the action name that to be invoked once the transition is triggered. A transition may be followed by an argument and guard. This will be discussed later in the SMC syntax and use section. The subway turnstile state diagram shown in Figure 3 can be described as follows:

- Turnstile is initially in the **Locked** state.
- When a **Coin event** occurs in the **Locked state** (a user inserts a coin), the turnstile transitions to the **Unlocked state** and the **Unlock action** is performed,

- While in the **Unlocked state**, if the user passes through the turnstile, hence, the **Pass event** occurs, then the turnstile transitions back to the **Locked state** and the **Lock action** is called.
- If the **Pass event** occurs while the turnstile is in the **Locked state** (perhaps some violation is taken place in this case), then the turnstile remains in **the Locked state**; thus **loopback transition**, and the **Alarm action** is performed.
- If the **Coin event** occurs in the **Unlocked state** (the user inserts extra coin,), then the turnstile sticks in the **Unlocked state (loopback transition)** and the **Thankyou action** is performed.

The last two transitions are loopback transitions and they can be viewed as unnatural transitions. Because, the events (Coin and Pass, normal events) that triggers these transitions happens to occur in incorrect time or state.

State machine diagrams provide a powerful, clear and easy way to reason about the completion and behavior of a desired system, such that unexpected events are easily discovered and handled.

As another representation, the subway turnstile state diagram in Figure 3 can be also described as a state transition table STT, (Table 1) as follows [3, 4]:

Current State	Event	Next state	Action
Locked	Coin	Unlocked	Unlock
Locked	Pass	Locked	Alarm
Unlocked	Pass	Locked	Lock
Unlocked	Coin	Unlocked	Thankyou

Table 1: State transition table

An STT representation is a data structure table [3, 4]. It provides clear and concise information, such that it can be easily interpreted and understood. The 16 words in the

Table 1 contain all the logic of the in question FSM. As it is shows, each row depicts a transition. The first row, for example, can be interpreted as follows: *“If we are in the Locked state and get a coin event, we go to the Unlocked state and invoke the unlock function.”*

2.2 Implementing Finite State Machines

There are several ways to implement finite state machines [3, 4, and 5]. The common way is to use nested switch case statements or If-then-else statement. State pattern is another effective choice to implement FSMs [3, 4, and 5].

2.2.1 Implementing Finite State Machines Using Nested Switch Case Statements or IF Then Else Statements

The following Table 2 shows [3, 4] the implementation of FSM in Figure 3 using nested switch case statements.

<pre> package subway.turnstile; public class Turnstile { private State state = State.LOCKED; private TurnstileActions turnstileAction; public Turnstile(TurnstileActions action) { turnstileAction = action; } public void Transition(Event e) { switch (state) { case LOCKED: switch (e) { case COIN: state = State.UNLOCKED; turnstileAction.Unlock(); break; case PASS: turnstileAction.Alarm(); break; } break; case UNLOCKED: switch (e) { case COIN: turnstileAction.Thankyou(); break; case PASS: state = State.LOCKED; turnstileAction.Lock(); break; } break; } } } </pre>	<pre> package subway.turnstile; public enum State { LOCKED, UNLOCKED } </pre>
	<pre> package subway.turnstile; public enum Event { PASS, COIN } </pre>
	<pre> package subway.turnstile; public interface TurnstileActions { void Lock(); void Unlock(); void Thankyou(); void Alarm(); } </pre>

Table 2: Nested switch case statements implementing turnstile FSM

The nested switch case statement in Turnstile class [3, 4] divides the code into “*four mutually exclusive cases*” each case matches a transition (event) in the FSM. That is, each case checks the event and accordingly transitions to the next state, and calls associated action. This approach is suitable and efficient when implementing simple finite state machines. In that the states and events are grouped in one place which eases visualizing and understanding the state machine logic. Nevertheless, this technique becomes complex for larger FSMs. Since nested switch case statements centralize checking for each state and event, the code gets lengthy and becomes difficult to read and maintain, thus, error prone. In the same manner [4, 5], this situation applies to the implementation of FSMs using if then else statements.

2.2.2 Implementing Finite State Machines Using State Pattern

State pattern “is to allow an object to change its behavior when its internal state changes” [6]. “The object will appear to change its class”. Simply stated, an object can have different states and its behavior depends on its present state, as well each state leads to a known next state [5]. Generally, state patter is used in any application where objects behavior is based on their states; for example, GUI applications, communication protocols, and electronic devices applications [5]. In essence, state patter is a suitable way to implement state machines since state machines are used to describe objects behavior according to their states. UML class diagram in the following Figure 4 shows the structure of the state pattern [6].

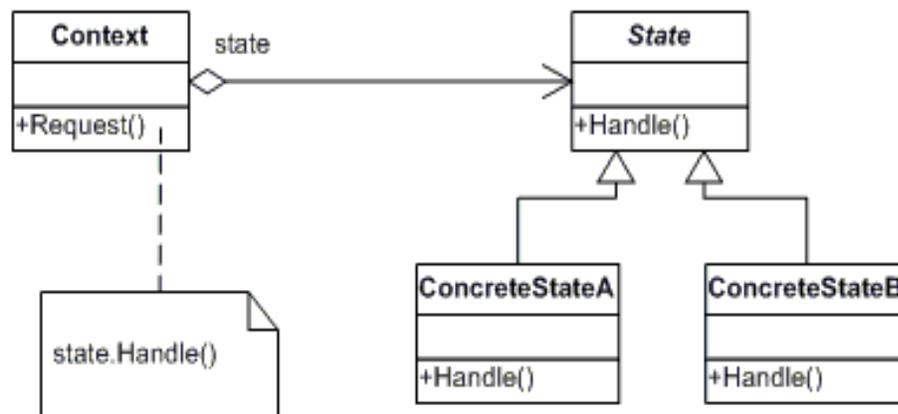


Figure 4: State pattern architecture

As it is shown in Figure 4, the architecture primarily consists of three classes [5, 6]:

- Context class: is a compartment holding all states and maintaining the current state (an instance of a ConcreteState class). The application will interact with this class.

- State class: is an abstract super class for concrete state classes. The reference of this class (“state”, as shown in the Figure) will be used in the context class to delegate and point to the current state.
- ConcreteState classes: implement the behavior associated with states of Context.

To put it simple, in state pattern each object state is represented by a concrete class which eases adding new states and extending existing ones. State pattern conforms to the “*Open-Closed Principle (OCP)*; that is, software entities should be open for extension but closed for modification” [1]. Besides, it can be easily seen that state pattern takes advantage of polymorphism [7].

Figure 5 shows [3] the use of state pattern to implement subway turnstile state machine example described previously in Figure 3 **Error! Reference source not found..**

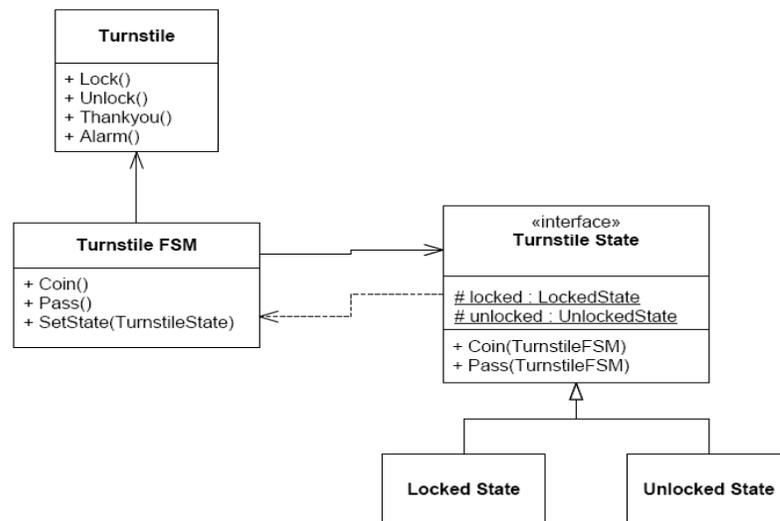


Figure 5: Turnstile FSM in state pattern

As it is shown in the Figure 5, Turnstile class implements the actions for Turnstile FSM (Lock, Unlock, Alarm, and Thankyou). This, in terms of the SMC concepts, is the application class;

<AppClass> (mentioned earlier in the introduction section, that a programmer should write to interact with the SMC generated code (FSM)). **However, In SMC the application class does not inherit from the TurnstileFSM class, as will be illustrated subsequently.** TurnstileFSM class implements the transitions (coin, pass) and it contains a reference to the generated TurnstileState interface. The reference will be used to delegate TurnstileFSM methods to their matches in TurnstileState. TurnstileFSM class corresponds to the context class in state pattern structure. **In terms of SMC; however, TurnstileFSM class is divided into two classes. That is, the generated context class that consists of transitions methods and it is used to interact with the application class <AppClass>, and the FSMContext class which is not generated but used by SMC “compiler”,** as will be discussed later. The TurnstileState interface acts as a place holder (it is equivalent to the abstract class named (State) in **Error! Reference source not found.** that shows the architecture of state pattern. Finally, there are two generated subclasses TurnstileState: LockedState and UnlockedState. Those correspond to the concrete classes in state design pattern. For example, if the FSM is in the Locked state, then TurnstileFSM will point at the LockedState and if a coin event occurs, the coin method in TurnstileFSM will be invoked this will delegate to the coin method of TurnstileState interface, which in turn will cause passing down to the coin method of LockedState. This method will invoke setState () method on the TurnstileFSM class and then will call the Unlock method on TurnstileFSM class which TurnstileFSM class inherits from Turnstile class.

2.3 SMC Motivation

State pattern reduces the complexity of implementing various states of an object such that it employs the separation of behavior and the logic of state machines [3]. In Figure 5, all the logic is contained in the TurnstileState hierarchy and all the behavior is contained in the Turnstile

hierarchy. So, such implementation is easily extended, as if there is a need to add a new class without modifying existing code. The behavior can be preserved and the logic can be changed separately and vice versa. On the contrary, the disadvantages of state design pattern are, the logic of the state machine is deployed in several classes (states), so that it cannot be visualized easily. Also, writing a class for each state is a hard and repetitive work.

Overcoming these obstacles and taking the advantage of state pattern efficiency [1], was the motivation to develop SMC. That is to say, first SMC describes state machine in one place, (.sm file), so the logic of FSM can be easily visualized. Second, SMC generates state pattern classes, so there is no need to manually write a class for every state. What is more, Since SMC generates code that interacts with a written application class <AppClass>, then, there is no need to modify or maintain the generated code. The main concern is to write and maintain the application class, as well as the SMC textual file (.sm file). The .sm file is simple in that it represents a one or more map; in other words, a state transition table. If SMC file contains multiple maps, then each map is basically a state transition table.

Charles W. Rapp, the creator and owner of SMC SourceForge project [1], is currently developing and maintaining SMC project along with other contributors. Charles's SMC was derived from Bob Martine's original state machine compiler. Charles W. Rapp has added many features to Bob Martin's initial state machine compiler; to enumerate some, (arguments, transition guards, push/pop transitions and default transitions). Since then several versions of SMC have been released. Next chapter introduces the syntax and use of SMC.

Chapter 3

SMC: Use and Syntax

3.1 SMC in Nutshell

On the whole, the following steps summarize how to use SMC in order to generate state pattern classes for state machines [1]:

1. Write the .sm file that describes the state machine.
2. Compile the file by running SMC to generate state pattern classes.
3. Write the application class <AppClass> to implement the actions.
4. Interact with the generated code (FSM) by calling transitions methods on the generated context class.
5. There is no need to change programmer's written code or to inherit any generated state machine classes.

Recall the state diagram for subway turnstile introduced in Figure 3 which is shown below, too [3]. It will be used to elaborate the syntax and layout of SMC file.

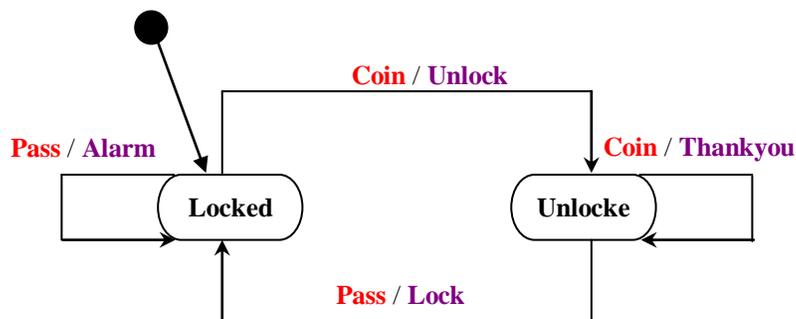


Figure 6: (Figure 3 Repeated) Simple turnstile FSM diagram

SMC syntax is simple, such that it resembles a state transition table STT; for example, the following Table 3 shows Locked state along with its transitions, next state and actions.

```
Locked
{
    Coin Unlocked {Unlock ();}
    Pass nil {Alarm ();}
}
```

Table 3: Simple SMC state syntax

This SMC code represents a state definition which includes its transitions, next state and actions. The state definition is placed inside a map block. Here, Locked represents the current state, Coin is the transition, Unlocked is the next state and Unlock () is the action which is placed inside the braces. As well Pass is another transition defined in Locked state, “nil” is the keyword used to indicate that this transition is loopback transition; hence, the end state is the current state (Locked state). Finally, Alarm () is the action.

The following tables, Table 4 and Table 5 [1, 4] demonstrate the basic syntax of the SMC textual file and how it interacts with the application class. Table 5 has more details explaining SMC file.

```

%{
}

Verbatim code section

%}

%class TurnstileAppClass Application class name

%package turnstile Package name

%access package Accessibility level

%start TurnstileFSM::Locked Start state

%map TurnstileFSM Map name
%%
    Locked {
        coin Unlocked { unlock(); }
        pass nil { alarm(); }
    }

    Unlocked {
        pass Locked { lock(); }
        coin nil { thankyou(); }
    }
%%
// BLUE color for keywords and symbols of SMC
// RED color for transitions names
// PURPLE color for actions names
// BLACK color for user defined names (e.g maps, states, etc)
// GREEN color for comments
}

/*
 * this is the application class that should be written,
 * interacts with FSM,
 * defines and instantiates FSM
 * implements action methods,
 * calls transition methods
 */
public class TurnstileAppClass implements TurnstileActions {
    // define the generated context class (FSM) instance
    TurnstileAppClassContext turnstileFSM;

    TurnstileActions _actions;

    public TurnstileAppClass (TurnstileActions actions) {
        // instantiate the generated context class
        turnstileFSM = new TurnstileAppClassContext (this);
        _actions = actions;
    }
    /*
     * transitions (events) methods,
     * direct transitions calls to a specific part of,
     * application if needed
     */
    public void coin() {
        turnstileFSM.coin();
    }
    public void pass() {
        turnstileFSM.pass();
    }
}
/*
 * actions methods,
 * implement or delegate actions
 */
public void lock() {
    _actions.lock();
}
public void unlock() {
    _actions.unlock();
}
public void alarm() {
    _actions.alarm();
}
public void thankyou() {
    _actions.thankyou();
}
}

```

Map block resembles state transition table (STT)

Table 4: SMC file syntax with application class

```

%o{ //=====> delimiter start of verbatim code section
    // this section is for copy rights and any other comments,
    // It will be literally copied to the generated source code
    // It is demarcated by %o{ and %o}

    /* comment support */
    // this is also another comment
%o} //=====> delimiter end of verbatim code section

/*
 * specify the application class that will interact with
 * the FSM by using the keyword %oclass
 * this file works only for the TurnstileAppClass class
 * TurnstileAppClass is the application class (AppClass) declared here
 * that should be written to define actions methods for FSM actions
 */
%oclass TurnstileApp Class

/*
 * specify the package name for the generated code using
 * %opackage keyword
 * the package is the same as AppClass class package
 * here turnstile is the package name
 */
%opackage turnstile

/*
 * specify the accessibility level using %oaccess keyword
 * works only for Java and C#
 * here the accessibility level is package
 */
%oaccess package

/*
 * specify the start state using %ostart keyword
 * followed by {map name}::{start state name}
 * here TurnstileFSM is the map name,
 * and Locked is the start state
 */
%ostart TurnstileFSM::Locked

/*
 * %omap keyword specifies the map name (FSM name)
 * states (or related states) are grouped into maps
 * %omap block starts with the %o%o and ends with %o%o delimiters
 * here TurnstileFSM is the map name
 */
%omap TurnstileFSM
%o%o //=====> delimiter start of map
    // basically a transition table
    // Locked state
    Locked {
        coin Unlocked { unblock(); }
        pass nil { alarm(); }
    }
    // Unlocked state
    Unlocked {
        pass Locked { lock(); }
        coin nil { thankyou(); }
    }
%o%o //=====> delimiter end of map

/*
 * the name of this file should be appended by (.sm) suffix
 * the name of the generated file that contains state pattern classes
 * is derived from the name of this file.
 * the class name in the generated file is derived from the AppClass
 * name declared in this file (TurnstileAppClass)
 * if generating code for java, then
 * this file name should match the application class name, as follows
 * TurnstileAppClass.sm
 * the { } are not required if they contain single statement,
 * yet for clarity it is strongly recommended, as within java or c syntax.
 */

```

Table 5: SMC file syntax with explanation comments

The verbatim block may only be used once in the .sm file and it must be placed at the top of the file before any SMC statement; however, comments may be used before the verbatim section.

Additionally [1], one can import namespaces or classes in the .sm file using **%import** keyword placed at the top of the file; for example, **%import java.awt.event.*** (without the semicolon at the end of line). Also, fully qualified class names are accepted for some target languages within **%class** keyword; for instance, **%class com.MyProject.AppClass** (for Java, C# and VB.net) and **%class::MyProject::AppClass** (for C++ and Tcl). Details on various SMC syntax and keywords will be discussed subsequently.

3.2 SMC Requirements

(The SMC 5_0_2 version released in January 14, 2008 is used in this report [1]). SMC is a java based tool [1]. Therefore, SMC requires Java 1.5.0 (JRE-standard edition) or later to be installed and “javac”, “java” and “jar” are in the PATH environment variable. Then, SMC can be downloaded, as open source software from <http://smc.sourceforge.net>. SMC package contains, among others, the executable Smc.jar (“SMC’s Compiler”) and statemap.* (SMC’s libraries), the “*” refers to multiple target programming languages endings; to mention some, statemap.jar for Java and statemap.h for C++ and statemap.dll for VB.Net. The statemap library is used only by SMC generated code. Programmer’s hand-written code (specifically AppClass – application class) has no use of this library. The full path to the ../Smc/bin (which contains smc.jar) should be added to the PATH environment variable, and also the full path to statemap may be added to the CLASSPATH environment. The SMC installation is quite simple such that it needs only to extract SMC zipped package and place it in the preferred directory taken in the account adding the full paths mentioned above. Up to date, SMC has no Eclipse plug in. Next section introduces how to compile SMC’s .sm file and various command line options.

3.3 SMC Compiling

SMC's command line basically takes the following arguments [1]:

Java -jar smc.jar -{target language} -{options} {sm file name}.sm

The options allow specifying some control on the generated code. To illustrate:

- **smc.jar**: represents the SMC “compiler”, so its directory should be specified
- **{sm file name}.sm** : **the (.sm) file** must be supplied as the last argument.
- – **{target language}**: one should specify which target language to generated code for.

Since SMC supports generating code in several programming languages, one of the following languages (SMC.V.5_1_0) should be specified:

-c: generates C code.

-c++: generates C++ code.

-csharp: generates C# code.

-groovy: generates Groovy code.

-java: generates Java code.

-lua: generates Lua code.

-objc: generates Objective-C code.

-perl: generates Perl code.

-python: generates Python code.

-ruby: generates Ruby code.

-tcl: generates [incr Tcl] code.

-vb: generates VB.net code.

-php: generates PHP code.

-scala: generates Scala code.

-{options}:

- **-table:** generates an HTML table describing the FSM.
- **-graph:** generates GraphViz.dot file.
- **-glevel:** (used only combining with **-graph** option) determines the details representation that will appear in the GraphViz diagram. It takes Integer value from 0 for least details to 2 for most details as follows:
 - glevel 0:** is for least detail; generates state names, transition names and pop transition nodes only.
 - glevel 1:** generates all of the **-glevel 0** plus transition guards and transition actions.
 - glevel 2:** generates all of the **-glevel 0** and **-glevel 1** plus state entry and exit actions, transition parameters, pop transition arguments and transition action arguments.
- **-sync:** (used only with the **-java**, **-groovy**, **-vb** and **-csharp**) causes to add synchronized keyword to the transition methods declarations in the context class as for **Java** and **Groovy** implementations. For **VB.Net** and **C#** implementation, **-sync** causes to encapsulate the transition method's body in Synclock Me, End Synclock block, and lock(this) block respectively. Hence, **-sync** insures thread safety in these specific supported languages.
- **-help:** prints SMC's command line options.
- **-version:** outputs the version of SMC.
- **-verbose:** produces verbose messages during compilation.
- **-suffix:** allows to specify the suffix for the generated file name than the default one.
- **-noex:** causes not to generate exception handling code. (Used only with the **-c++**); Generates assert() rather than thrown exceptions in case there are unhandled errors; for instance, when an action yields a transition from within a transition. SMC, even when

using **-noex**, still generates try/catch/rethrow blocks in order to protect the FSM against application thrown exceptions.

- **-nocatch**: causes not to generate try/catch/rethrow blocks.
- **-cast**: (used only with the **-c++**), sets casting operator. The allowed C++ cast operators are: `dynamic_cast` (default), `static_cast` and `reinterpret_cast`. For instance **-cast** can be used as follows: `-cast static_cast`
- **-d**: determines the output directory for the generated code. By default the generated code is the same directory as the `(.sm)` file. When specifying a particular directory, the directory must be accessible from the current working directory and writeable.
- **-headerd**: (used only with the **-c**, **-c++** and **-objc** options.), specifies output directory for `(.h)` generated files. If neither **-d** nor **-headerd** is specified, then `(.h)` files are placed in the same directory as the `(.sm)` file.
- **-g**: causes to add debugging output to the generated code. Yet, this output is not generated unless turned on at run time. To do so; for example in Java: `_state_machine.setDebugFlag (true);` where `_state_machine` is the FSM instance. By default the debugging output is placed in the standard error; for instance, in Java: `(System.err)`, but this can be changed as follows: `_state_machine.setDebugStream(java.io.PrintStream).`
- **-nostreams**: (used only with the **-c++** and **-g** options.), causes the generated code not to use `IOStreams` for debugging output, but the application must provide subroutines or macros to handle such situations.
- **-serial**: used when persisting SMC's finite state machines, as will be discussed in the persistence section.
- **-reflect**: used with reflection, further details are presented in the reflection section.

- **-return:** causes SMC not to exit.

3.4 States and Transitions

A state and transition names must have the form "[A-Za-z_][A-Za-z0-9_]*" [1] (see Appendix A for SMC EBNF grammar [1]). SMC defines several types of transitions [1], namely, standard or simple, jump, loopback, push, and pop and default transitions. A transition definition includes four parts with some exceptions according to the transition type, generally the parts are:

- The transition's name, which may be followed by a comma-separated argument in parentheses.
- The transition may have a guard.
- The next or end state of the transition.
- The transition may include actions which represent the first mating between FSM and application class <AppClass> that defines actions methods.

3.4.1 Simple Transition

Simple transitions are standard transitions that lead from current state to new next state. Figure 7 shows a snippet of turnstile FSM describing a simple transition and

Table 6 shows the equivalent SMC code.



Figure 7: Simple transition

```

Locked
{
    Coin Unlocked {Unlock ( ) ;}
}
  
```

Table 6: SMC code for simple transitions in Figure 7

3.4.2 Jump Transition

In May 20, 2008, SMC new version (SMC_5_1_0) has been released with some updates, including Jump transition [1]. The Jump transition is similar to the Simple transition in syntax and use and it is intended to be used in augmented transition network (ATN) [1]. The augmented transition network (ATN) [1, 8] is an FSM used to parse sentences in natural language processing. ATNs include transition guards, push/pop transitions, default transitions and backtracking transition [1]. In essence, SMC concepts come from ATN [1]. Further information on ATNs can be found in [1, 8].

3.4.3 Loopback Transition

Loopback transitions [1] are simply used when there is a need to remain in the same state. The keyword “nil” is used to indicate that the next state is the current state; thus, loopback transition. Alternatively, the current state name can be used instead of “nil”, except in the “Default state” where “nil” keyword is required; the reason for that is because “Default” as a keyword, is used to define a “Default” state and transition, so it cannot be used to indicate the loopback transition. More on Default state and transitions will be discussed in the Default state and transitions sections. (The next planned SMCv.6.0.0 will distinguish between using nil and state name for loop backing, more on this at the end of this section). Figure 8 shows a portion of turnstile FSM representing a simple loopback transition and Table 7 shows the equivalent SMC code.

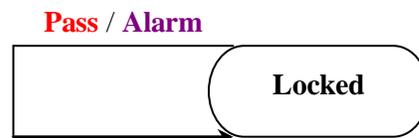


Figure 8: Loopback transition

```

Locked
{
    Pass nil {Alarm () ;}
}

// or

Locked
{
    Pass Locked {Alarm () ;}
}

```

Table 7:

Loopback Transition

It should be mentioned that SMC currently makes no deference when using the keyword “nil” or the state name. However, the next planned release of SMC (SMC.v. 6.0.0) will handle loopback transitions differently. That is, SMC will support two loopback transitions: internal and external. The internal loopback transition is defined by the keyword “nil” and when used it causes not to execute the state Entry and Exit actions. The external loopback transition is defined when using the state name itself instead of “nil”. In this case, Entry and Exit actions are executed.

3.4.4 Push/Pop Transition

Since SMC allows defining multiple maps; then, there is a need to transition among them and return back. Push and pop transitions are intended for this purpose. That is to say, push transition can be used to move to a certain state across maps; conversely, pop transition cause to return to the state that is prior to the push transition. When pushing to a certain state, the actions associated with push transition are performed, and then the moving to the target state is taken. In the contrary, pop transition does not define a next state, obviously because it returns to the last state before pushing, but it contains a transition name as an argument which will be performed in the

returned state. Also, this argument may contain additional arguments (this will be explained in the next example), as well as, pop transition may have actions. For one reason, this is why a state may receive an event for which there is no transition defined in it, and so here where the benefit of “Default” state and transitions comes; that is, to recover from such situations. “Default” state and “Default” transitions are introduced afterwards in the “Default” state and transitions section. Push and pop transitions can also be used to move between states in the same map. As well a standard or plain transition can be used to move to other maps. Yet, it is not recommended since the intention of pushing and popping comes from the need of multiple maps each of which contains related states; thus, simplifying the logic of the state machine and resembling simplicity of state transition tables; on the other hand, corresponding to the calling techniques of subroutine programming. Consider the following Figure 9 [3] that shows extended turnstile FSM with a violation state and diagnostic mode.

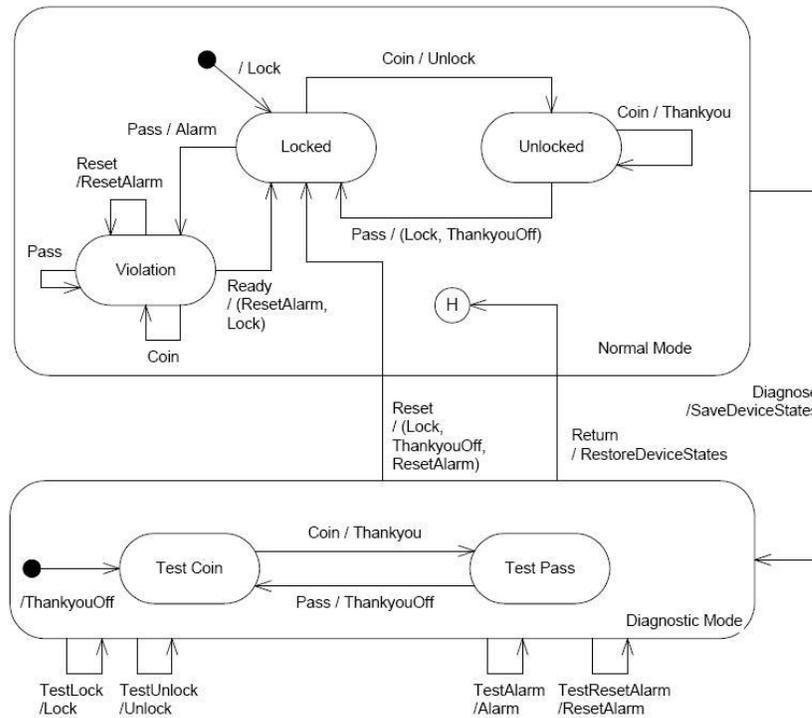


Figure 9: Turnstile FSM with violation state and diagnostic mode

The violation state is defined and transitioned to when the pass event occurs while in the Locked state. The only way to exit the violation state is through Ready transition which indicates the end of the violation state. Otherwise, any other transition will be loop backed, as the case with Pass, Coin and Reset transitions. The purpose of Reset transition is to allow turning the alarm off while working on the turnstile. Also, the diagnostic state (maintenance mode) is added in order to allow maintenance and checking for the turnstile functionality. As it can be seen in Figure 9, there are two super states, Normal Mode and Diagnostic Mode. Each super state contains and groups some related sub states. That is, Normal states (normal mode or operations) of turnstile are placed into one super state which is referred to as Normal Mode. Diagnostic states (maintenance mode or operations) are grouped in one super state called Diagnostic Mode. In terms of SMC concepts, each super state corresponds to a certain map; thus, there are two maps, Normal and Diagnostic.

Normal map is the main one; hence, in SMC coding, it should be the only one that identifies the start state of the whole FSM. In terms of state pattern, super states correspond to the abstract classes or “interfaces”, as it was discussed in the implementing state machines using state pattern section. To compare, abstract classes can only be instantiated using its sub classes and super states can only be entered as part of a sub state. The question is how to transition between these two super states- (or maps in SMC?) In the Figure 9, the Diagnostic transition leaves from the Normal Mode super state itself, but not from a certain sub state in it. To illustrate, the Diagnostic transition can be triggered from any sub state in the Normal mode. Super states simplify state diagrams such that, as in this case, there is no need to draw or define the same Diagnostic transition for each sub state in the Normal Mode to transition to the Diagnostic Mode. When leaving the Normal Mode, the action SaveDeviceStates is called to preserve the states in the Normal Mode. Also, when entering the Diagnostic Mode, one of its sub states must be entered. In this case, the start state TestCoin, as it is shown in the Figure, is indicated by the initial pseudo state; thus, TestCoin is entered. Similarly, in SMC when pushing to a certain map using push transition, the Push transition must include the state specifying where to start in the target map. On the other hand, returning to the Normal Mode, as the case in the Figure 9, can be made in several ways. First, the Reset event puts the turnstile back into the Locked state; performs the actions of Locking, setting the Alarm to off and the Thankyou light off. Second, the Return event restores the states of the Normal Mode, and then enters the History pseudo state indicated by a small circle with H inside. That is, the sub state in the Normal Mode that must be entered is the sub state that was last exited. Likewise, in SMC pop transition is used to return back to the last state prior to pushing. The following Table 8 introduces, and then the illustration shows, how push and pop transitions can be used with multiple maps in SMC code.

```

%{
%}
%class TurnstileAppClass
%package turnstile
%access package
%start TurnstileNormalMode::Locked
%map TurnstileNormalMode
%% // start of TurnstileNormalMode
Locked
{
    Coin Unlocked {Unlock();}
    Pass Violation {Alarm();}
    DONE nil {TestedTime (time);}
    FAILED (reason: String) nil {Report (reason);}
    Go push (Unlocked) {}
}
Unlocked
{
    Pass
        Locked
        {
            Lock();
            ThankyouOff();
        }

    Coin nil {Thankyou();}
    Return pop (StartDiagnostic) {}
}
Violation
{
    Coin nil {}
    Pass nil {}
    Reset nil {ResetAlarm();}
    Ready
        Locked
        {
            ResetAlarm();
            Lock();
        }
}
Default
{
    Default [ctx.is Time (name, time)] Locked / push (TurnstileDiagnosticMode::TestCoin) {GetStatus ();}
}
%% // end of TurnstileNormalMode

%map TurnstileDiagnosticMode
%% // start of TurnstileDiagnosticMode
TestCoin
{
    Coin TestPass {Thankyou();}
    TestLock nil {Lock();}
    TestAlarm nil {Alarm();}
    TestResetAlarm nil {ResetAlarm();}
}
TestPass
{
    Pass TestCoin {ThankyouOff();}
    TestUnlock nil {Unlock();}
    TestAlarm nil {Alarm();}
    TestResetAlarm nil {ResetAlarm();}
}
Default
{
    Default [ctx.is Tested ()] pop (Done) {}
    Default pop (FAILED, reason) {}
}
%% // end of TurnstileDiagnosticMode

```

Table 8: Multiple maps with push and pop transitions

In the Table 8 example two maps are defined and push/pop transitions are used to move between them, as well to move inside the same map. Table 8 is not precisely corresponding to the state diagram in Figure 9 because Table 8 defines more states and transitions (“Defaults”) than Figure 9, that is for the seek of illustrating push and pop transitions, as well as for keeping the state diagram more simple. Push transition in Table 8 once is defined as follows:

Default

```
{  
    Default [ctxt.isTime (name, time)] Locked/push (TurnstileDiagnosticMode::TestCoin) {GetStatus () ;}  
}
```

In this case, Push transition is defined within the “Default” transition inside “Default” state. The reason for this is that moving to the Diagnostic Mode can be made from any sub state (Locked, Unlocked and Violation). Also, the “Default” transition is guarded; consequently push transition will not be taken until a certain time comes as specified in the guard. The meaning of push transition: **Locked/push (TurnstileDiagnosticMode::TestCoin) {GetStatus () ;}** is this:

1. Transition to the Locked state.
2. Perform the Locked state entry actions.
3. Push to the TurnstileDiagnosticMode::TestCoin state
4. Execute the TurnstileDiagnosticMode::TestCoin state entry actions. TestCoin is now the current state and its transitions are handled.
5. When TurnstileDiagnosticMode issues pop transition, the Exit actions of the popping state will be executed, then the control will return to the Locked state without executing its Entry actions. The Locked state is now the current state and its transitions will be handled.

When all tests in the Diagnostic Mode; likely, are done, TurnstileDiagnosticMode issues a pop transition to return back to the Locked state in the Normal Mode from which the push transition was issued. The pop transition in Table 8 once is defined as follows:

Default

```
{  
    Default [ctxt.isTested ( )] pop (DONE)  
    Default pop (FAILED, reason)  
}
```

Pop transition here is also triggered from The “Default” transition inside “Default” state. The purpose of this is that returning back to the Normal Mode can be made at any time and sub state in the Diagnostic Mode, as well as “Default” state’s guarded- “Default”-transition and its unguarded-“Default”-transition each has the less precedence respectively among other transitions types. Hence in this example, “Default” transition; therefore pop transition, has two versions, guarded and unguarded. If the guarded one returns true, then the control will return back to the Locked state from which pushing was made and the DONE transition will be taken in the Locked state. Yet, if the guarded “Default” transition returns false, then the second “Default” transition will be taken and the FAILED transition will be performed in the Locked state. Note, the pop transitions arguments consists of the transition names (DONE and FAILED) and passed arguments (reason) in FAILED. Additionally, push and pop transitions in Table 8 are also, for explanation reasons, defined to move in the same map, TurnstileNormalMode. In the Locked state push transition is introduced as follows:

```
Go push (Unlocked) { }
```

This syntax is the initial syntax for push transition but it is compatible with the new one introduced previously. The push transition in this case causes to transition to the Unlocked state. In the Unlocked state pop transition is defined as follows:

Return pop (StartDiagnostic) { }

Pop transition will cause to return back to the Locked state and handle the StartDiagnostic transition. However, Locked state does not have StartDiagnostic transition; therefore, the control will be taken to the “Default” state which has guarded “Default” transition which; in turn; contains push transition mentioned above. Similarly, push/pop transitions can be defined in other sub states in the Normal Mode.

3.4.5 Transition Actions

Recall that actions are the first coupling between SMC’s FSM and the application class [1]. Thus, actions must be member methods in the application class and accessible by the generated code [1]. In other words, actions methods in the programmer application class must be public in order to be accessed by generated code. It is possible to declare action methods as private; however, each state class will be maintained manually to access actions method. Also, with every modification such as deleting or renaming a state, the application class have to be updated which is tedious and time consuming work. Generally, the reason of this is because SMC intended to work with several target programming languages. Particularly, in Java actions methods can have package level accessibility when the application class and the generated code are in the same Java package. In essence, it is important to handle and define actions and actions methods effectively in both, SMC file and application class. In SMC, Actions are placed after the end state of a transition such that they must be enclosed between a pair of braces “{ } “. An action definition consists of the action name (corresponds to a method name in the application class) followed by argument list. The argument list is enclosed in parenthesis “()” which may be either empty or contains a comma separated list. Argument list may contain integers (including, decimal, octal or hexadecimal and both positive or negative), floating points, strings enclosed in double quotes, constants, method calls and transition arguments (transition arguments is discussed in the next

section). When calling a method on application class as an action argument, the method name should be suffixed by “ctxt.”. The suffix “ctxt.” is used only inside argument lists and transition guards. Figure 10 show a simple use of action arguments and Table 9 shows the equivalent SMC code.

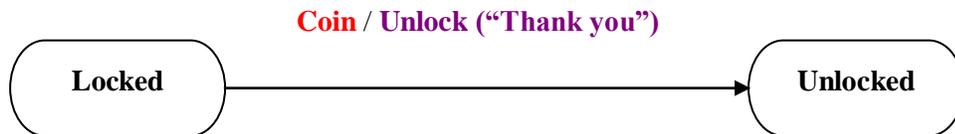


Figure 10: Simple transition action argument

```

Locked
{
    Coin Unlocked {Unlock (“Thank you”) ;}
}

```

Table 9: SMC simple transition action

3.4.6 Transition Arguments

Transitions may have argument list enclosed in parenthesis “()” [1]. When defining multiple unique-guarded transitions with the same name, they must have the same argument list in order to be considered as the same transition. That is to say, if the argument list is not the same for two guarded transitions with the same name, then these two transitions are not the same even though they share the same name. Transition argument list may include the same types as with transition actions. In the next section, transitions argument list and transitions guards will be discussed in more details and examples.

3.4.7 Transition Guards

Guards are conditions or Boolean expressions that must evaluate to true in order for transitions to be taken. Guards are placed after a transition name and its argument (if exists) [1]. And, they are placed inside a pair of square brackets “[]”. The allowed argument types for a transition guard are the same as with transition action arguments discussed earlier in the transition actions section.

Guards’ expression may contain logical and comparison operators, such as (&&, ||, ==, >, etc) and nested expression. SMC copies guard conditions literally into the generated code. If a guard expression contains a method call on the application class, then the method’s name in SMC coding must be preceded by the prefix “ctxt.” As well, the method invocation may take argument list. Besides, a state may contain multiple transitions with the same name and argument list, but distinctive guards. In this case, SMC will check these transitions in the same given order, top to bottom, except for the unguarded version which is always taken if all corresponding guarded transitions evaluate to false. As a consequence, the guarded transitions order is important; specifically, for multiple guards that evaluate to true for the same event. In other words, the first top guarded transition will be evaluated and if true, it will be taken. Yet, if this guard condition evaluates to false, then SMC will check for one of the choices in the following order:

- 1- If the state defines other guarded transitions with the same name and arguments, then these will be evaluated orderly from top to bottom till one evaluates to true. Else, if this is not the case, then
- 2- If the state defines unguarded transition with the same name and arguments, then this transition will be taken, but not evaluated since, obviously, it does not have a guard. Else, if it is not the case with those two options, then
- 3- The default transition logic is considered. That is, if a “Default” state is defined, then it will be checked for a corresponding transition to be evaluated if guarded or taken if not

guarded. Else, if the “Default” state is not defined, then SMC will check for a “Default” transition defined in the considered state. The “Default transition, in SMC rules, takes no arguments, but may have a guard. Therefore, because of this, and also because its name is “Default” –but not a certain name, it serves as a back up for all transitions in that state. As a result, a very careful consideration should be taken in account when dealing with “Default” state and “Default” transitions (the “Default” state and transitions are discussed in the next section). The “*following ordered list*” summarizes the transitions definitions precedence :

1. *Guarded transition*
2. *Unguarded transition*
3. *The Default state's guarded definition.*
4. *The Default state's unguarded definition.*
5. *The current state's guarded Default transition.*
6. *The current state's unguarded Default transition.*
7. *The Default state's guarded Default transition.*
8. *The Default state's unguarded Default transition”*

To make it clear, consider the following Figure 11 [1] which shows the use of transition arguments, guards, precedence, and also actions arguments. Table 10 the equivalent SMC code.

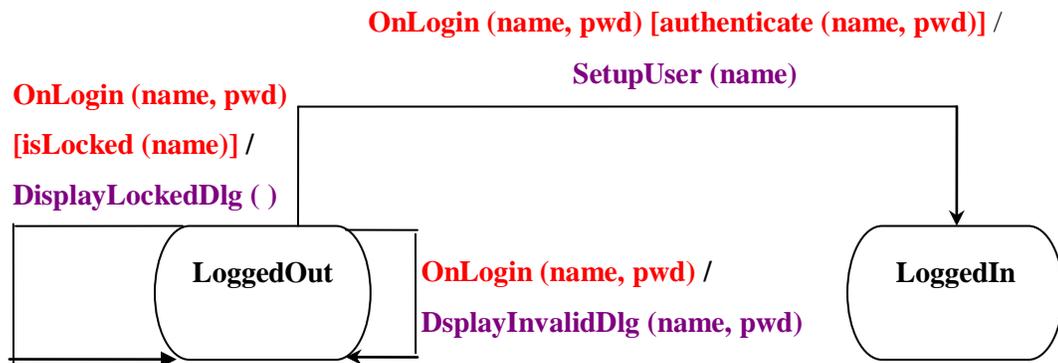


Figure 11: Transition with arguments, guards, and actions with transition arguments

```

LoggedOut
{

    OnLogin (username: String, password: String)
        [ctxt.isLocked (username)]
            nil {DisplayLockedDlg () ;}

    OnLogin (username: String, password: String)
        [ctxt.authenticate (username, password)]
            LoggedIn {SetupUser (username) ;}

    OnLogin (username: String, password: String)
            nil {DisplayInvalidDlg (username, password) ;}

}
  
```

Table 10: SMC equivalent code of Figure 11

The state diagram shown in Figure 11 has two states, LoggedIn and LoggedOut. There are three versions of the same transition. That is, three transitions with the same name and arguments. In other words, onLogin transition is a multiple transition leading only from the LoggedOut state to the LoggedIn state if its guard is true (the username and password are authenticated); otherwise, if the username is locked, then remain in the LoggedOut state, (loopback transition), and perform the action displayLockedDlg (). The third unguarded onLogin transition version is also a loopback transition and is taken when each of the other two guarded transitions evaluate to the false. In this case, the displayInvalidDlg (username, password) action will be invoked. So, same-guarded-transitions are taken first in order from top to bottom, if any evaluates to true, ignore other transitions and if all false, take unguarded version; if there is no unguarded transitions, perform the default transitions methodology as it is mentioned at the beginning of this section. As it can be seen in Table 10, the method invocation is preceded by “ctxt”. Likewise, the transitions define an argument which in turn is used as the actions argument list.

3.4.8 Default State and Transition

Going back to the transitions types presented earlier, (standard, jump, loopback, and push/pop); a “Default” transition is another kind and is discussed in this section along with a “Default” state [1]. SMC does not support state inheritance; instead, it adds the default state and transition as an equivalent technique. That is, allowing transitions to have virtual definitions; in other words, if a transition is not defined in a certain state, then default state or default transition will take place; thus, this is similar to calling or overriding super classes’ methods. Two next sections demonstrate “Defaults” technique with some examples.

3.4.9 Default State

Any map may contain a Default state [1]. The Default state is defined by the keyword “Default”. Consider the following example in Figure 12 , which shows the use of the Default state in turnstile FSM.

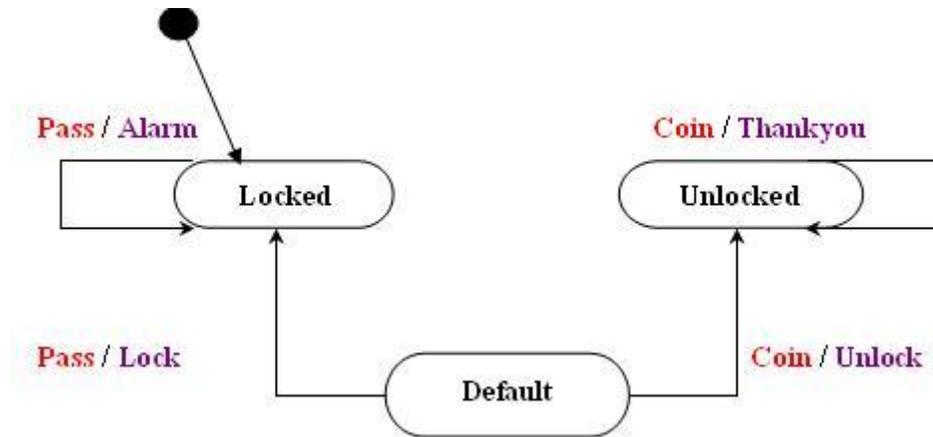


Figure 12: Default state

Since the Default state is not a “real” state and cannot be transitioned to, drawing it as it is shown in the Figure 12 is not appropriate. The intention of showing it in the diagram is for explanation purpose. Note that the coin transition is not defined in the Locked state and if the Locked state receives this transition to transition to the Unlocked state, then the coin transition in the Default state will be invoked to transition to the Unlocked state and the Unlock action will be called. Similarly, the pass transition is not defined in the Unlocked state; therefore, the pass transition in the Default state will be taken. The state diagram in Figure 12 can be coded in SMC language as follows:

```

Locked
{
    Pass nil {Alarm ( ) ;}
}

Unlocked
{
    Coin nil {Thankyou ( ) ;}
}

Default
{
    Coin Unlocked {Unlock ( ) ;}
    Pass Locked {Lock ( ) ;}
}

```

Table 11: Defining Default state in SMC language

Here, the Default state is used to define the “normal” transitions, coin and pass of Locked and Unlocked state respectively. So, it overrides the default behavior of turnstile FSM. In essence, “Default” state transitions, as normal transitions, may have arguments and guards. Thus, the “Default” state may have multiple guarded and one unguarded definition for the same transition. Also, the “Default” state may have a default transition using the same keyword “Default”. Default transition is discussed in the following section.

3.4.10 Default Transition

There are two ways to define default transitions [1]. The first one is that the “Default” state which defines transitions, including a “Default” transition, that serve as a “fallback” for all other states; hence, “Default” state transitions can be considered as default transitions since they are placed in

the “Default” state. That is, if there is a missing transition definition (as it is shown in Figure 12 and Table 11) in a specific state and that transition is used, then SMC will invoke the corresponding Default state’s transition if defined. The second way is a “Default” transition itself. The “Default” transition can be placed in any state including Default state using “Default” keyword to back up all transitions in that state. Deem the following example:

```

Locked
{
    Coin Unlocked {Unlock () ;}

    Default nil {LockedError () ;}
}

Unlocked
{
    Pass Locked {Lock () ;}

    Default Unlocked {UnlockedError () ;}
}

Default
{
    Coin nil {Thankyou () ;}
    Pass nil {Alarm () ;}
}

```

Table 12: Defining Default transitions and Default state in SMC file

In the Table 12, Default transitions and Default state are defined. First, let’s look at the Default transitions. In the Locked state there are two transitions, coin and Default. If any transition than coin occurs in Locked state, then its Default transition will be taken regardless of what a

transition is. Likewise, in the Unlocked state if any transition than pass occurs in it, then its Default transition will be performed. However, since the Default state is also defined in Table 12, so it has a higher precedence than the Default transitions defined within states. To illustrate, assume that Locked state receives pass transition which is not defined in it, then the “Default” transition defined in Locked state will not be taken because the Default state defines the pass transition which will transition to the same state, as “nil” keyword indicates and Alarm action will be called, instead of LockedError action. As it can be seen in Table 12, the use of “nil” is compulsory in the “Default” state, but in other states, the state name, optionally can be used instead, as the case in Table 12 in the “Default” transition of Unlocked state where the state name (Unlocked) is used instead on “nil”. **(The future release of SMC will distinguish between using nil and state name, see Loopback transition section for further details)**. Since default transitions defined within a certain state using “Default” keyword can be triggered if there is any undefined transition in that state, so they may not have argument list, but they may take a guard. In fact, placing a Default transition in the Default state itself will cause all transitions in the FSM to be handled regardless of being undefined.

Indeed, Default state and transitions prevent system crashing. They can be thought as exceptions handling or recovering technique. Since defining “Default” states and “Default“ transition are optional, it is the case that SMC may encounter undefined transitions. Therefore, SMC will throw a “Transition Undefined” exception.

3.4.11 State Entry and Exit Actions

Entry and Exit actions are those that performed whenever entering and exiting a state, respectively [1]. However, in SMC state’s Entry and Exit actions execution relays on the type of transition being taken, *“as shown in the following table”* [1]:

Transition Type	Execute "From" State's Exit Actions?	Execute "To" State's Entry Actions?
Simple Transition	Yes	Yes
Loopback Transition	No	No
Push Transition	No	Yes
Pop Transition	Yes	No

Table 13: Entry and Exit actions execution and transitions types dependency

Consider the following state diagram with Entry and Exit actions, Figure 13 , and also the equivalent SMC code in Table 14.

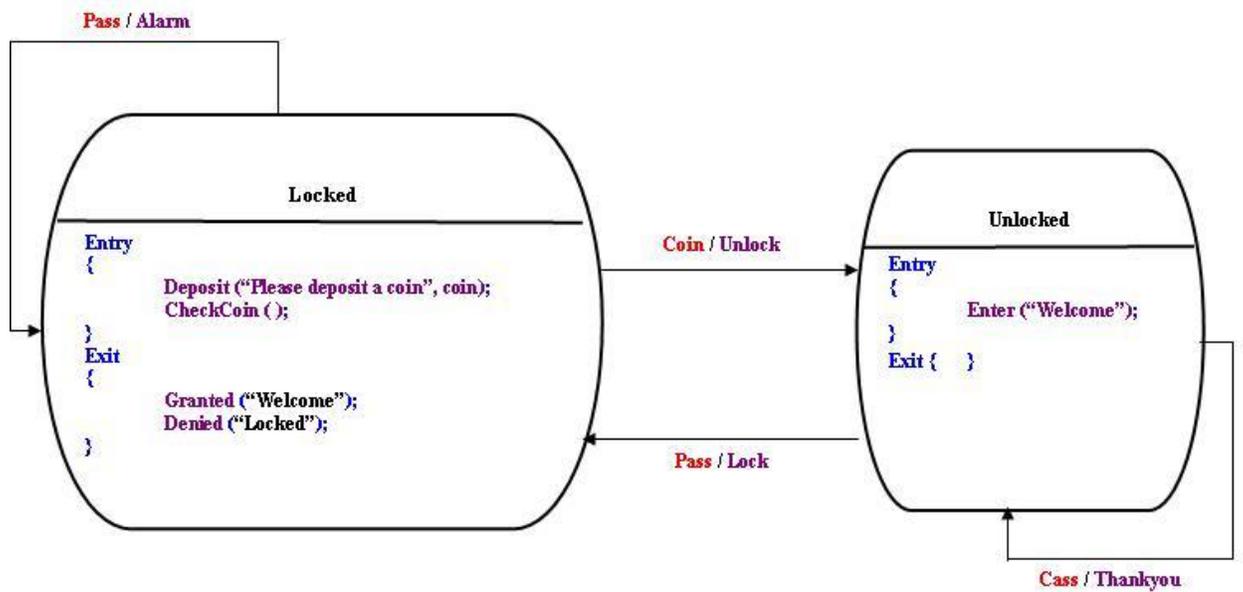


Figure 13: Turnstile State diagram with Entry and Exit actions

```

Locked
  Entry
  {
    Deposit (“Please deposit a coin”, coin);
    CheckCoin ();
  }
  Exit
  {
    Granted (“Welcome”);
    Denied (“Locked”);
  }
{
  Coin Unlocked {Unlock ();}
  Pass nil {Alarm ();}
}
Unlocked
  Entry
  {
    Enter (“Welcome”);
  }
  Exit { }
{
  Pass Locked {Lock ();}
  Coin nil {Thankyou ();}
}

```

Table 14:Entry and Exit actions in SMC

Note that Entry and Exit actions are placed immediately after a state name and before the starting “{“ of the state.

3.4.12 Transitions Issue Transitions

SMC does not allow issuing a transition from within an action and it throws an exception if an action does issue a transition [1]. This is clearly because while in the transition, an object is no more in any specific state. It is in between states where actions occur. Therefore, it is not

reasonable to issue a transition from a transition. However, it is the case where there is a need to take different transition depending on an action results. In such certain circumstances it is possible to place the action in the transition guard; thereby, the next transition that will be taken relays on that associated transition guard (in this case associated action) result. As another solution, the action can be placed in a state's entry action list from which the action can issue the transition depending on its result. That is, an object while in state entry actions is already in a known state and a different transition can be taken, therefore. However, this technique can lead to face some problems. When using this way, it is strongly recommended that the transition is issued by the last entry action to guaranty that all state's entry actions are executed. Also, a different way is to use timers and transitions queues to issue a transition within a transition. Issuing a transition within a current transition may lead to generate a buggy code since it requires knowing what actions are invoked and in what order.

Chapter 4

SMC Pattern and Generation

4.1 SMC Pattern

As mentioned before, SMC follows state pattern and generates state pattern classes (see implementing state machines using state pattern section). Consider the following Figure 14 [1] which shows SMC pattern with Turnstile example..

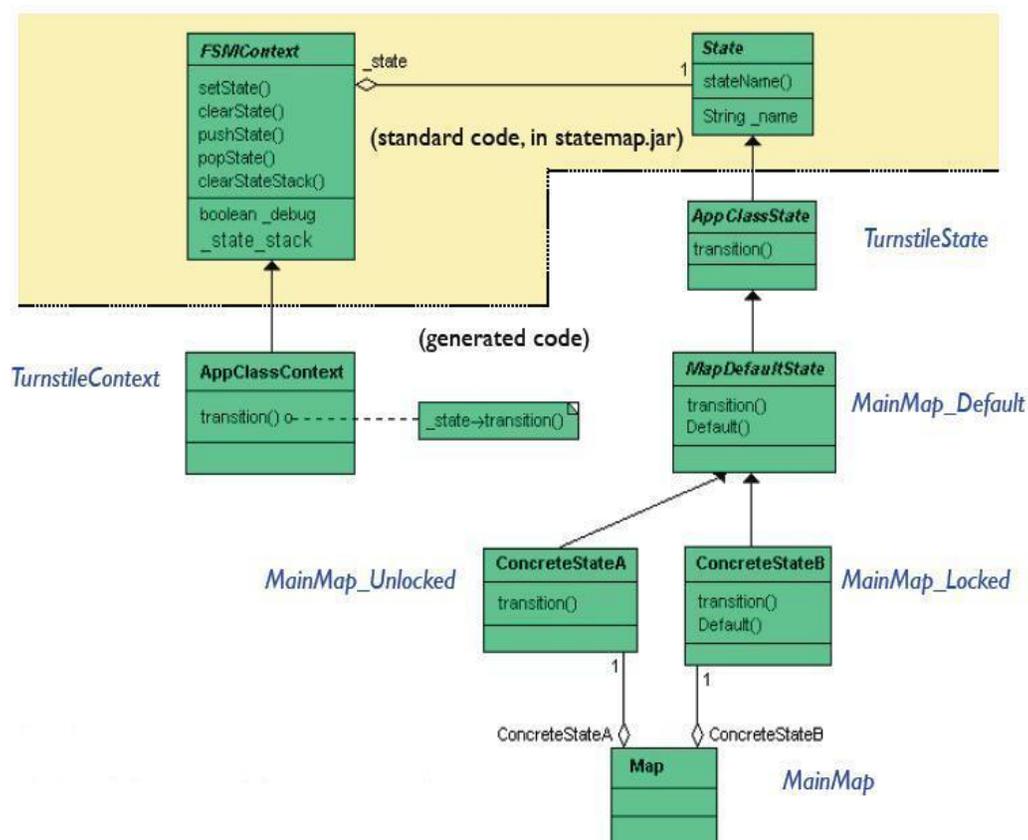


Figure 14: SMC pattern

Since SMC supports use of multiple maps, push/pop transitions, default states and default transitions, its state pattern is extended from the original state design pattern (see Figure 4). That

is, as it can be seen in the Figure 14, the Context class is divided into two classes: an abstract FSMContext class: this is not generated but used by SMC compiler and <AppClass>Context class: this is the generated class that interacts with AppClass – application class (recall AppClass is the application class that should be hand written) such that the <AppClass>Context class (generated context class) defines transitions methods which are called by AppClass – application class. The FSMContext class maintains the current state, as well as the state stack which used to track pushing and popping states. Also, FSMContext class defines methods for setting the state, and pushing/popping states. The <AppClass>Context is a subclass of FSMContext which defines a getState () method and maintains a reference to <AppClass- application class> object. The getState () method returns the current state as a <AppClass – application class>State object, but not a State object; therefore, this method is defined in this class instead of FSMContext class. Likewise SMC; unlike, state pattern which has an abstract State class and ConcreteStates subclasses, has four levels: State, <AppClass>State, Map Default state and concrete states. The reasons for this extension levels are that to support SMC’s default state and transitions. To illustrate, first <AppClass>State declares a virtual method (abstract) for each transition in the state machine. These transitions methods invoke <AppClass> State’s Default transition method. Furthermore, the map default class contains Default state’s transitions. And, each concrete state is a subclass of its map’s default state class. The state class defines methods that implement state machine transitions. The map class and concrete state classes are singletons; in other words, the map class declares one instance of its concrete state classes and it has no methods. That is, the map class is a compartment that aggregates a map’s state instance into one location. Specifically, SMC generates several classes, yet there is only one instance of each state class and only one class (<AppClass>Context) is instantiated for each <AppClass> class instance. Thus, SMC uses less run time space.

SMC is not directly related to UML or Harel state machines [1]. That is, SMC uses multiple maps and pushing and popping states to simplify the state machine logic, in that it follows subroutine calls technique. On the other hand, UML groups states into super states to accomplish a similar concept.

4.2 SMC Generated Code

The following SMC hand-written file (TurnstileAppClass.sm) corresponds to the Turnstile state diagram shown in Figure 3 **Error! Reference source not found.** and is the same code as the file showing previously in Table 4 and Table 5, but without verbatim code and comments:

```
%class TurnstileAppClass

%package turnstile

%access package

%start TurnstileFSM::Locked
%map TurnstileFSM
%%
    Locked
    {
        Coin Unlocked {Unlock(); }
        Pass nil {Alarm(); }
    }
    Unlocked
    {
        Pass Locked {Lock(); }
        Coin nil {Thankyou(); }
    }
%%
```

Table 15: SMC code describes simple Turnstile FSM

Compiling this SMC file using Java as a target language, the Following Java code (state pattern classes) is generated. The code is generated in one Java file named TurnstileAppClassContext.java:

```

package turnstile;

/* package */final class TurnstileAppClassContext extends
statemap.FSMContext {
    //-----
    // Member methods.
    //

    public TurnstileAppClassContext(TurnstileAppClass owner) {
        super();

        _owner = owner;
        setState(TurnstileFSM.Locked);
        TurnstileFSM.Locked.Entry(this);
    }

    public TurnstileAppClassContext(TurnstileAppClass owner,
        TurnstileAppClassState initState) {
        super();
        _owner = owner;
        setState(initState);
        initState.Entry(this);
    }

    public void coin() {
        _transition = "coin";
        getState().coin(this);
        _transition = "";
        return;
    }

    public void pass() {
        _transition = "pass";
        getState().pass(this);
        _transition = "";
    }
}

```

```

        return;
    }

    public TurnstileAppClassState getState()
        throws statemap.StateUndefinedException {
        if (_state == null) {
            throw (new statemap.StateUndefinedException());
        }

        return ((TurnstileAppClassState) _state);
    }

    protected TurnstileAppClass getOwner() {
        return (_owner);
    }

    public void setOwner(TurnstileAppClass owner) {
        if (owner == null) {
            throw (new NullPointerException("null owner"));
        } else {
            _owner = owner;
        }

        return;
    }

    //-----
    // Member data.
    //

    transient private TurnstileAppClass _owner;

    //-----
    // Inner classes.
    //

```

```

    public static abstract class TurnstileAppClassState extends
statemap.State {
        //-----
        // Member methods.

        protected TurnstileAppClassState(String name, int id) {
            super(name, id);
        }

        protected void Entry(TurnstileAppClassContext context) {
        }

        protected void Exit(TurnstileAppClassContext context) {
        }

        protected void coin(TurnstileAppClassContext context) {
            Default(context);
        }

        protected void pass(TurnstileAppClassContext context) {
            Default(context);
        }

        protected void Default(TurnstileAppClassContext context) {
            throw (new
statemap.TransitionUndefinedException("State: "
+ context.getState().getName() + ", Transition: "
                                + context.getTransition()));
        }

        //-----
        // Member data.
    }

```

```

/* package */ static abstract class TurnstileFSM {
    //-----
    // Member methods.
    //
    //-----
    // Member data.
    //
    //-----
    // Constants.
    //
    public static final
TurnstileFSM_Default.TurnstileFSM_Locked Locked = new
TurnstileFSM_Default.TurnstileFSM_Locked(
        "TurnstileFSM.Locked", 0);
    public static final
TurnstileFSM_Default.TurnstileFSM_Unlocked Unlocked = new
TurnstileFSM_Default.TurnstileFSM_Unlocked(
        "TurnstileFSM.Unlocked", 1);
    private static final TurnstileFSM_Default Default = new
TurnstileFSM_Default(
        "TurnstileFSM.Default", -1);
}

    protected static class TurnstileFSM_Default extends
TurnstileAppClassState {
    //-----
    // Member methods.
    //
    protected TurnstileFSM_Default(String name, int id) {
        super(name, id);
    }

    //-----
    // Inner classe.

```

```

//

private static final class TurnstileFSM_Locked extends
    TurnstileFSM_Default {
    //-----
    // Member methods.
    //

    private TurnstileFSM_Locked(String name, int id) {
        Super(name, id);
    }

    protected void coin(TurnstileAppClassContext context) {
        TurnstileAppClass ctxt = context.getOwner();

        (context.getState()).Exit(context);
        context.clearState();

        try {
            ctxt.unlock();
        } finally {
            context.setState(TurnstileFSM.Unlocked);
            (context.getState()).Entry(context);
        }

        return;
    }

    protected void pass(TurnstileAppClassContext context) {
        TurnstileAppClass ctxt = context.getOwner();

TurnstileAppClassState endState = context.getState();

        context.clearState();
        try {
            ctxt.alarm();
        } finally {

```

```

        context.setState(endState);
    }
    return;
}

//-----
// Member data.
//
}

private static final class TurnstileFSM_Unlocked extends
    TurnstileFSM_Default {
//-----
// Member methods.
//

private TurnstileFSM_Unlocked(String name, int id) {
    Super(name, id);
}

protected void coin(TurnstileAppClassContext context) {
    TurnstileAppClass ctxt = context.getOwner();

TurnstileAppClassState endState = context.getState();

    context.clearState();
    try {
        ctxt.thankyou();
    } finally {
        context.setState(endState);
    }
    return;
}

protected void pass(TurnstileAppClassContext context) {

```

```

        TurnstileAppClass ctxt = context.getOwner();

        (context.getState()).Exit(context);
        context.clearState();
        try {
            ctxt.lock();
        } finally {
            context.setState(TurnstileFSM.Locked);
            (context.getState()).Entry(context);
        }
        return;
    }

    //-----
    // Member data.
    //
}

//-----
// Member data.
//
}
}

```

Table 16: Generated code: TurnstileAppClassContext.java

The following Figures: Figure 15, Figure 16, Figure 17 show class diagrams for the generated code. That is, the diagrams are generated according to the generated code in Table 16. These diagrams are generated using Eclipse [9] and Omondo [10] UML plug in for Eclipse.

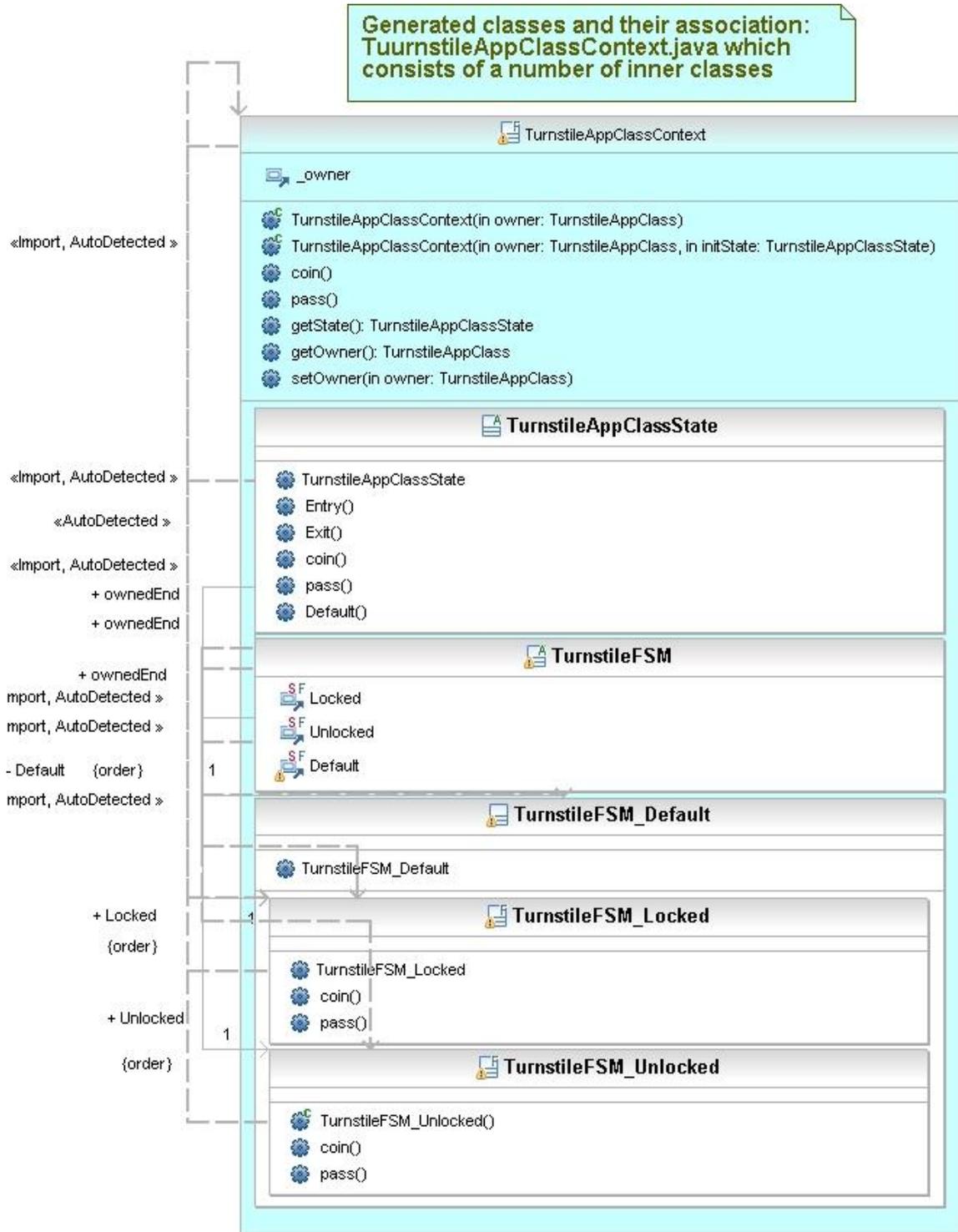


Figure 15: Class diagram for the generated code in Table 16

As it is stated previously, the generated code is grouped in one Java file called `TurnstileAppClassContext.java`. This file consists of `TurnstileAppClassContext` class (Blue color in the Figure 15) along with a number of inner classes (White color) as it is shown in the Figure 15. The following Figure 16 shows the dependency between the generated classes (Blue color), classes in `statemap` library (SMC's library, not generated – in Yellow color), and the programmer's hand written code (application class – Pink color).

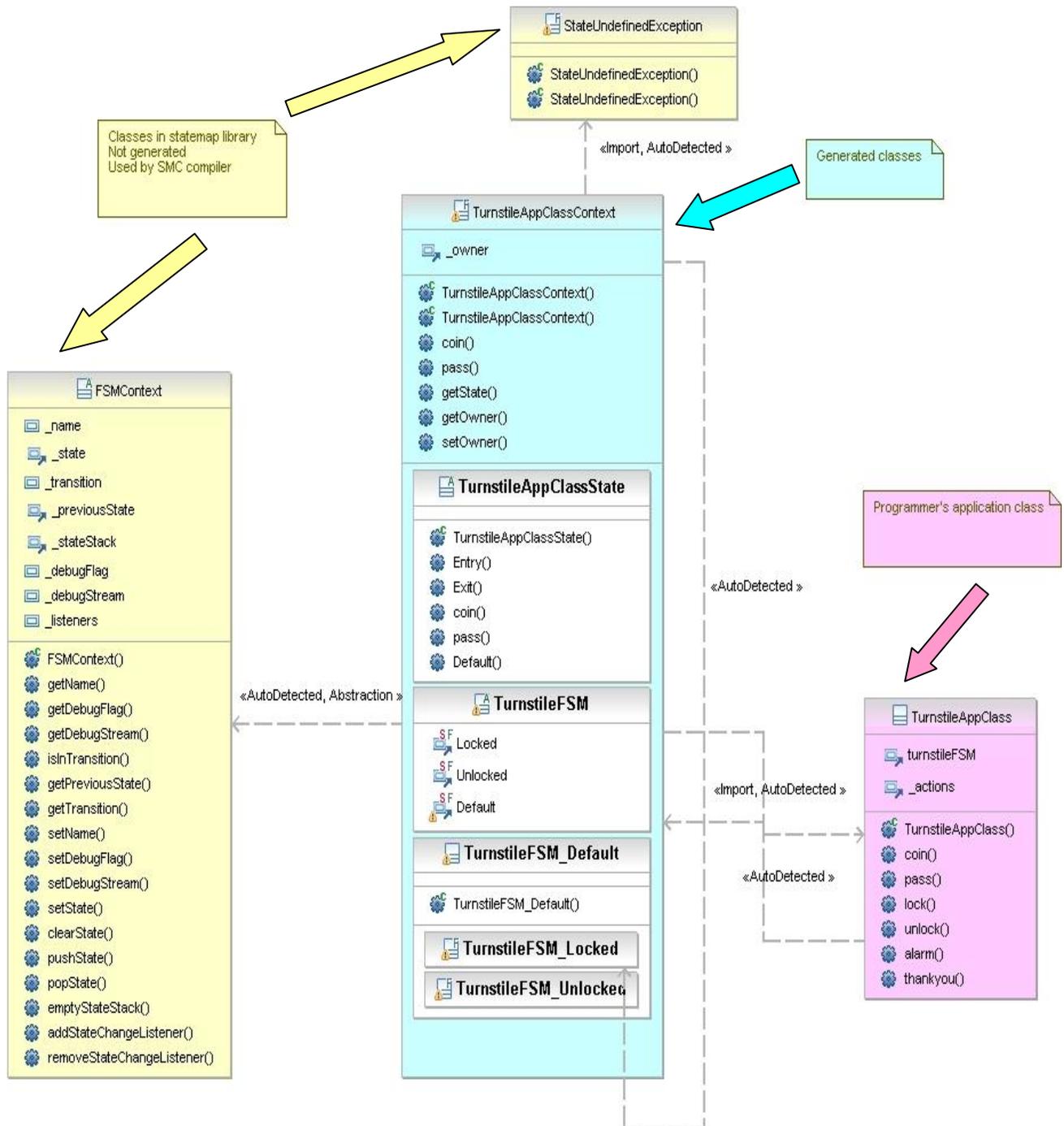


Figure 16: Dependency between generated classes, statemap library and application class, (corresponds to the generated code in Table 16)

Now consider Figure 17, “a big picture”, which shows the complete view of the SMC’s written file, programmers written application class, generated classes, statemap library classes that SMC compiler (smc.jar) uses to interact with the generated classes. Pink color is for manual code; that is, application class (TurnstileAppClass.java), SMC (.sm) files (TurnstileAppClass.sm) and an interface (TurnstileActions.java). Blue color is for the generated classes; that is TurnstileAppClassContext which acts as a compartment that groups inner classes. Yellow color is for the classes in the statemap library. The figure also shows the dependency and association among these various classes, as well as SMC compiler (Red color).

4.3 Generating GraphViz Dot File

In addition to the generated state pattern classes, SMC generates a GraphViz.dot file [1]. GraphViz [11] is open source graph visualization software. It takes descriptions of graphs in a textual file and generates diagrams in several formats. More details and downloading information can be obtained at [11]. In essence [1], by using SMC command line options, `-graph` and `-glevel`, SMC generates GraphViz.dot file from which, by using GraphViz, a diagram describing FSM can be drawn. The `-glevel <0, 1 or 2>` options determine the amount of details that will be shown in the diagram, as it is mentioned in the SMC command line options section. Compiling the SMC file shown in Table 15 with `-graph` and `-glevel`, the following Figures: Figure 18, Figure 19 and Figure 20 show the generated diagrams with `-glevel 0`, `-glevel 1`, `-glevel 2` details respectively.

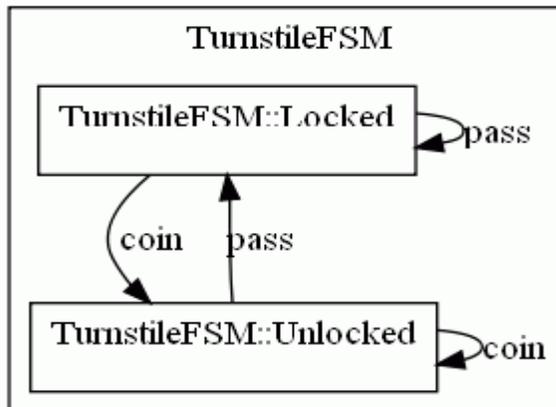


Figure 18: Turnstile diagrams with least details (-glevel 0)

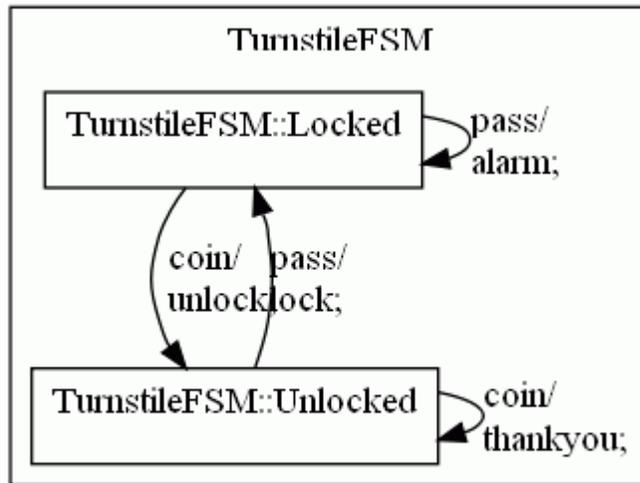


Figure 19: Turnstile diagrams with -plevel 1

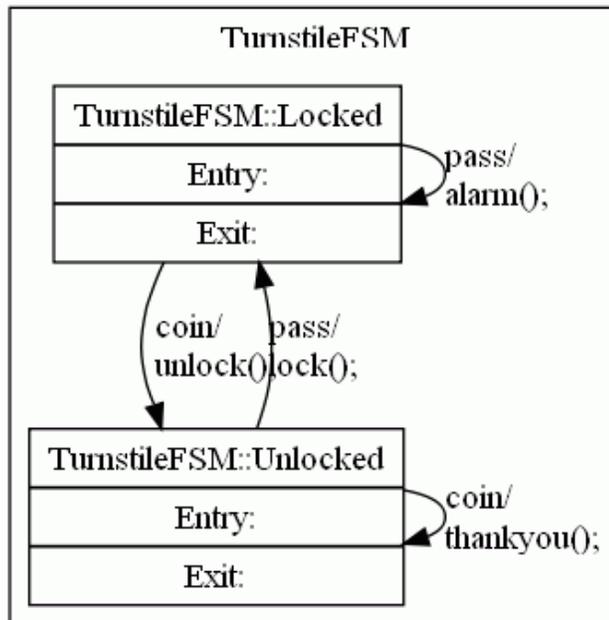


Figure 20: Turnstile diagrams with -plevel 2

Since in this example, there are no transition arguments, pop transition arguments and state entry and exit actions, so they are not represented in Figure 20. But, consider the following diagram which is generated for the SMC file in Table 8.

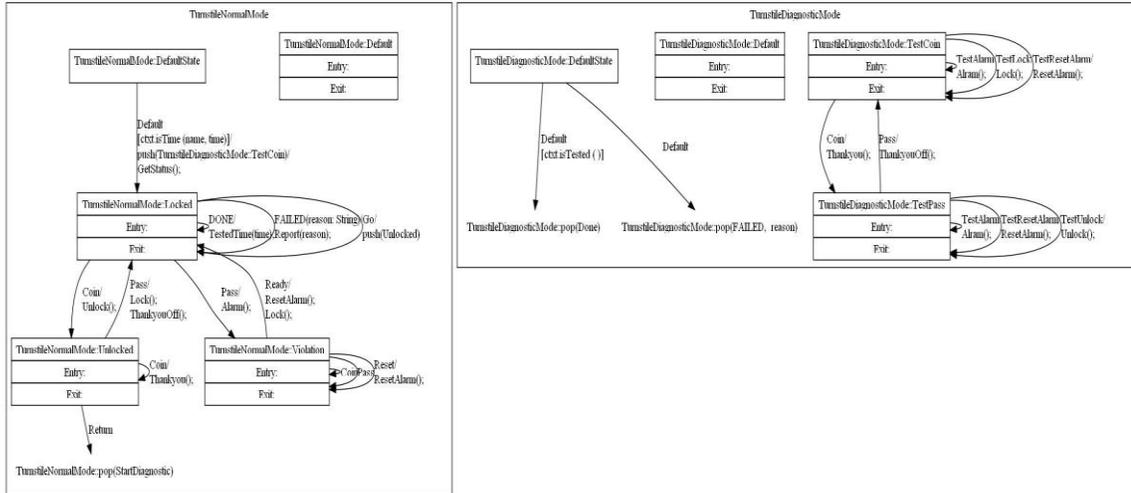


Figure 21: Turnstile diagram with multiple maps

In the above diagram there are two maps: `TurnstileNormalMode` and `TurnstileDiagnosticMode` map. SMC [1] when generating GraphViz dot file, it represents each map in a separate sub graph, and also as it is shown in all previous GraphViz drawings, all states names are represented by a full qualified named; that is a map name and state name format (`<map name>::<state name>`). The reason for this is that GraphViz has a single global namespace whereas SMC has a separate namespace for each map. In other words, for instance, `NormalMap::Start` and `DiagnosticMap::Start` in one SMC file yields two different Start states. Yet, GraphViz sees these two Start states to be the same node; thus, it draws only one node named Start. This is due to nodes in GraphViz exist in the same global namespace. Therefore, to resolve this situation, SMC when generating GraphViz dot file, uses the fully qualified state name.

4.4 Generating HTML Table

Using the command line option `-table [1]`, SMC generates an HTML table includes the actions for each state/transition pair, each state's entry and exit actions as it is shown in the following Table 17.

TurnstileNormalMode Finite State Machine											
State	Actions		Transition								
	Entry	Exit	Coin	DONE	FAILED (String)	Go	Pass	Ready	Reset	Return	Default
Locked			Unlocked { Unlock(); }	Locked { TestedTime(t ime); }	Locked { Report(reason); }	push(TurnstileNormalM ode::Locked) { }	Violation { Alarm(); }				
Unlocked			Unlocked { Thankyou(); }				Locked { Lock(); ThankyouOff(); }			pop(StartDi agnostic) { }	
Violation			Violation { }				Violation { }	Locked { ResetAlarm(); Lock(); }	Violation { ResetAlarm(); }		

TurnstileDiagnosticMode Finite State Machine									
State	Actions		Transition						
	Entry	Exit	Coin	Pass	TestAlarm	TestLock	TestResetAlarm	TestUnlock	Default
TestCoin			TestPass { Thankyou(); }		TestCoin { Alarm(); }	TestCoin { Lock(); }	TestCoin { ResetAlarm(); }		
TestPass				TestCoin { ThankyouOff(); }	TestPass { Alarm(); }		TestPass { ResetAlarm(); }	TestPass { Unlock(); }	

Table 17: Generated HTML table for SMC file in Table 8

4.5 Persistence

SMC's command line "-serial" is required to be used when persisting SMC's finite state machines [1]. Persistence is a way of storing, and then restoring data. In other words, writing/reading data in/from a resource. SMC generated finite state machine can be persisted via the instance of FSM that is defined in the application class, <AppClass> and the data to persist is the FSM current state and state stack. Regarding Java as a target language, SMC takes advantage of Java's object serialization. In this case, the application class should implement java.io.Serializable and the FSM instance (<AppClass>Context class object) should not be a transient. Serializing the <AppClass> instance will result to serializing FSM. That is, when the application class, <AppClass> is serialized, its associated <AppClass>Context class is serialized too; however the opposite way is not true. Also, when <AppClass>Context class is deserialized, the <AppClass> class reference is not.

4.6 State Change Notification

State change notification [1] is an important feature that is supported by SMC for some of the supported target programming languages. SMC employs Java Bean event notification and .Net event raising features to inform listeners when state change occurs. In essence, SMC does not generate event registration and listening code. Instead, the application class receives and maintains the events, then passes them to the FSM (<AppClass>Context class.). FSM; then, keeps track of the object's state.

4.7 Reflection

For reflection [1], SMC uses `-reflect` command line option for Java, C#, Perl, PHP, Python, Ruby, Tcl and VB.Net programming languages. By using reflection, SMC generates either a `getTransitions` methods or a `Transitions` property for C# and VB.Net. The returned value is a map represents transitions names. Transitions names are map keys with an integer value. For Java the method definition is: "*Public Map getTransitions()*" where map key is a String (transition name) and the value is an integer. Integer values are:

0: the transition is undefined in the current state.

1: the transition is defined in the current state.

2: the transition is undefined in the default state.

Knowing the current state's transitions is useful especially when developing interfaces such that some features in the interface can be activated or deactivated based in the current state. Particularly, calling `getState ()` method while in a transition is not possible because the state is not yet known. The `getPreviousState ()` can be called while transitioning to determine the previous state.

Chapter 5

Conclusion

5.1 Closing Points

SMC brings a great benefit of automatically implementing finite state machines in several programming languages, following state pattern. In the same way, SMC is simple in that its file is easy to write and understand. SMC's syntax is the same for all supported target languages. In fact, SMC is not a programming language or a real compiler, but it is "*a glorified macro generator*" [1]. That is to say, SMC does not compile its file such as guards or arguments, but instead it reads these literally and writes them out to the target language, depending on the specified language to find errors. Indeed, SMC decouples the implementation of the actions, from the FSM logic; that is, the state transition behavior which is implemented by SMC as methods in the generated context class. Thus, instantiating one instance of the generated context class and invoking its transitions method in the application class is only what needed to interact with the FSM. Also, SMC generates state pattern classes in one file consisting of several inner classes. Hence, it takes less run time space. For one thing, SMC's file considers a certain application class for a specific target language; therefore, SMC requires determining a target language in the command line option. As a result, this imposes limitation on the generated code such that the .sm file for a particular class and a specific language can not be reused in a different application or target language without certain changes. However, the advantages of targeting several programming languages and generating state pattern classes is more worthy than specifying a certain language in the command line or doing minor alerts in the code [1]. Put another way, SMC leverages implementation of FSM in several programming language using the power of

state pattern with the least impact to the target application; therefore, it tries to use the lowest common programming language constructs. Consequently, the complexity of the target programming language is centralized and hidden in the intended application class. Regarding Default state and Default transition, one can say that the use of the same keyword “Default” for the “Default” state and the “Default” transitions is “a wrong design decision”. Since SMC primarily and heavily relies on those Defaults and all states may have default transitions including the default state itself, this may lead SMC file to be difficult to follow, and also to differentiate between Default states and Default transitions, despite the indentation, and especially with large FSMs that make much use of default states and transitions, hence, error-prone. Furthermore, SMC [1] accepts only simple method calls; for example, “*object().name()*” call is not accepted due to the support of multiple languages. Further, Push/pop and simple transitions can be used interchangeably. That is to say, what can be done using push/pop transitions can also be done using simple transition. In other words, simple transition can be used in the same way and for the same purpose as push/pop transitions to move across maps.

References

- [1] Charles W. Rapp, “SMC SourceFroge project”: Home, What is New, SMC Overview, SMC Programmer’s Manual, FAQ, “downloadable” SMC Tutorial, SMC Project’s Latest News, SMC Forums: Help, Open Discussion. <http://smc.sourceforge.net>, Last updated May 20, 2008

- [2] Thomas, D. and Hunt, A.: “State Machines”, IEEE SOFTWARE November/December 2002, Vol: 19, Issue: 6, pgs: 10-12, ISSN: 0740-7459.

- [3] Martin Robert C.: “UML Tutorial: Finite State Machines”, Engineering Notebook Column, C++ Report, June 1998, <http://www.objectmentor.com/resources/articles/umlfsm.pdf>

- [4] Martin Robert C. and Micah Martin: “Agile principles, patterns, and practices in C#”, Prentice HallPub, July 20, 2006, Safari Books Online “2007, Pearson Education, Inc.” <http://proquestcombo.safaribooksonline.com.proxy.queensu.ca/0131857258>

- [5] Sarath: “Understanding State Pattern in C++”, 25 Jun 2006, http://www.codeproject.com/Kb/architecture/StatePatternBy_Sarath_.aspx

- [6] “State Design Pattern in C# and VB.NET”, dofactory.com, <http://www.dofactory.com/Patterns/PatternState.aspx>

- [7] Armstrong E.: “How to implement state-dependent behavior”, JavaWorld, 08/01/97, <http://www.javaworld.com/jw-08-1997/jw-08-stated.html>

- [8] WIKIPEDIA, The Free Encyclopedia, http://en.wikipedia.org/wiki/Augmented_transition_network

- [9] Eclipse SDK, version 3.3.2, <http://www.eclipse.org/platform>

- [10] Omondo UML plug in for Eclipse Europa, Release: EclipseUML 2007 Europa Free Edition for Eclipse 3.3, version 3.3.3.3.0.v20071210, <http://www.omondo.com>

- [11] GraphViz- Graph Visualization Software, <http://www.graphviz.org>

Appendix A

SMC EBNF Grammar

SMC EBNF are straightforward and easy to understand. The following description [1] introduces SMC EBNF:

```
FSM := source? start_state class_name header_file? include_file*
package_name* import* declare* access* map+
source := '%{' raw_code '%}'
start_state := '%start' word
class_name := '%class' word
header_file := '%header' raw_code_line
include_file := '%include' raw_code_line
package_name := '%package' word
import := '%import' raw_code_line
declare := '%declare' raw_code_line
access := '%access' raw_code_line
map := '%map' word '%%' states '%%'
states := word entry? exit? '{' transitions* '}'
entry := 'Entry {' actions* '}'
exit := 'Exit {' actions '}'
transitions := word transition_args? guard? next_state '{' actions '}'
transition_args := '(' parameters ')'
parameters := parameter |
              parameter ',' parameters
parameter := word ':' raw_code
guard := '[' raw_code ']'
next_state := word |
             'nil' |
             push_transition |
             pop_transition
```

```

push_transition := word '/' 'push(' word ')' |
                 'nil/push(' word ')' |
                 'push(' word ')'

pop_transition := 'pop' |
                 'pop(' word? ')' |
                 'pop(' word ',' pop_arguments* ')'

pop_arguments := raw_code |
                raw_code ',' pop_arguments

actions := dotnet_assignment |
           action |
           action actions

dotnet_assignment := word '=' raw_code ';'

action := word '(' arguments* ');'

arguments := raw_code |
            raw_code ',' arguments

word := [A-Za-z][A-Za-z0-9_]* |
        _[A-Za-z][A-Za-z0-9_]*
<div class="comment">// Reads in code verbatim until end-of-line is
reached.</div>
raw_code_line := .* '\n\r\f'

<div class="comment">// Read in code verbatim.</div>
raw_code := .*

<div class="comment">// Both the // and /* */ comment types are
supported.</div>
<div class="comment">// Note: SMC honors nested /* */ comments.</div>
comment1 := '/' .* '\n\r\f'
comment2 := '/*' .* '*/'

```