Spring Statemachine - Reference Documentation

1.0.0.M2

Janne Valkealahti Pivotal

Table of Contents

Preface

Concept of a state machine is most likely older that any of a reader of this reference documentation and definitely older than a Java language itself. Description of finite automate dates back to 1943 when gentlements Warren McCulloch and Walter Pitts wrote a paper about it. Later George H. Mealy presented a state machine concept in 1955 which is known as a Mealy Machine. A year later in 1956 Edward F. Moore presented another paper which is known as a Moore Machine. If you're ever read anything about state machines, names Mealy and Moore should have popped up at some point.

This reference documentations contains following parts.

Part I, "Introduction" introduction to this reference documentation

Part II, "Spring and Statemachine" describes the usage of Spring State Machine(SSM)

Part III, "State Machine Examples" more detailed state machine samples

Part IV, "FAQ" frequently ask questions

Part V, "Appendices" generic info about used material and state machines

Part I. Introduction

Spring Statemachine(SSM) is a framework for application developers to use traditional state machine concepts with Spring applications. SSM aims to provide following features:

- Easy to use flat one level state machine for simple use cases.
- Hierarchical state machine structure to ease complex state configuration.
- State machine regions to provide even more complex state configurations.
- Usage of triggers, transitions, guards and actions.
- Type safe configuration adapter.
- State machine event listeners.
- Spring IOC integration to associate beans with a state machine.

Before you continue it’s worth to go through appendices Section B.2, “Glossary” and Section B.3, “A State Machines Crash Course” to get a generic idea of what state machines are mostly because rest of a documentation expects reader to be fairly familiar with state machine concepts.

1. Requirements

Spring Statemachine 1.0.0.M2 is built and tested with JDK 7 and Spring Framework 4.1.6.RELEASE and doesn’t require any other dependencies outside of Spring Framework. Samples require spring-shell and spring-boot which pulls other dependencies beyond framework itself.

2. Background

State machines are powerful because behaviour is always guaranteed to be consistent and relatively easily debugged due to ways how operational rules are written in stone when machine is started. Idea is that your application is and may exist in a finite number of states and then something happens which takes your application from one state to the next. What will drive a state machine are triggers which are either based on events or timers.

It is much easier to design high level logic outside of your application and then interact with a state machine with a various different ways. You can simply interact with a state machine by sending event, listening what a state machine does or simply request a current state.

Traditionally state machines are added to a existing project when developer realizes that code base is starting to look like a plate full of spaghetti. Spaghetti code looks like never ending hierarchical structure of IFs, ELSEs and BREAK clauses and probably compiler should ask developer to go home when things are starting to look too complex.

3. Usage Scenarios

Project is a good candidate to use state machine if:

- Application or part of its structure can be represented as states.
- You want to split complex logic into smaller manageable tasks.
- Application is already suffering concurrency issues with i.e. something happening asynchronously.

You are already trying to implement a state machine if:

- Use of boolean flags or enums to model situations.
- Having variables which only have meaning for some part of your application lifecycle.
- Looping through if/else structure and checking if particular flag or enum is set and then making further exceptions what to do when certain combination of your flags and enums exists or doesn’t exist together.

Part II. Spring and Statemachine

This part of the reference documentation explains the core functionality that Spring Statemachine provides to any Spring based application.

Chapter 4, *Statemachine Configuration* describes the generic configuration support.

Chapter 5, *State Machine Factories* describes the generic state machine factory support.

Chapter 9, *Triggering Transitions* describes the use of triggers.

Chapter 10, *Listening State Machine Events* describes the use of state machine listeners.

Chapter 11, *Context Integration* describes the generic Spring application context support.

4. Statemachine Configuration

One of the common tasks when using a Statemachine is to design its runtime configuration. This chapter will focus on how Spring Statemachine is configured and how it leverages Spring's lightweight IoC containers to simplify the application internals to make it more manageable.

> **Note**
>
> Configuration examples in this section are not feature complete, i.e. you always need to have definitions of both states and transitions, otherwise state machine configuration would be ill-formed. We have simply made code snippets less verbose by leaving other needed parts away.

4.1 Configuring States

We'll get into more complex configuration examples a bit later but lets first start with a something simple. For most simple state machine you just use `EnumStateMachineConfigurerAdapter` and define possible states, choose initial and optional end state.

```
@Configuration
@EnableStateMachine
public static class Config1 extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1)
                .end(States.SF)
                .states(EnumSet.allOf(States.class));
    }

}
```

4.2 Configuring Hierarchical States

Hierarchical states can be defined by using multiple `withStates()` calls where `parent()` can be used to indicate that these particular states are sub-states of some other state.

```
@Configuration
@EnableStateMachine
public static class Config2 extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1)
                .state(States.S1)
                .and()
                .withStates()
                    .parent(States.S1)
                    .initial(States.S2)
                    .state(States.S2);
    }

}
```

4.3 Configuring Regions

There are no special configuration methods to mark a collection of states to be part of an orthogonal state. To put it simple, orthogonal state is created when same hierarchical state machine has multiple set of states each having a initial state. Because an individual state machine can only have one initial state, multiple initial states must mean that a specific state must have multiple independent regions.

```
@Configuration
@EnableStateMachine
public static class Config10
        extends EnumStateMachineConfigurerAdapter<States2, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States2, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States2.S1)
                .state(States2.S2)
                .and()
                .withStates()
                    .parent(States2.S2)
                    .initial(States2.S2I)
                    .state(States2.S21)
                    .end(States2.S2F)
                    .and()
                .withStates()
                    .parent(States2.S2)
                    .initial(States2.S3I)
                    .state(States2.S31)
                    .end(States2.S3F);
    }

}
```

4.4 Configuring Transitions

We support three different types of transitions, `external`, `internal` and `local`. Transitions are either triggered by a signal which is an event sent into a state machine or a timer.

```
@Configuration
@EnableStateMachine
public static class Config3 extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1)
                .states(EnumSet.allOf(States.class));
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source(States.S1).target(States.S2)
                .event(Events.E1)
                .and()
            .withInternal()
                .source(States.S2)
                .event(Events.E2)
                .and()
            .withLocal()
                .source(States.S2).target(States.S3)
                .event(Events.E3);
    }

}
```

4.5 Configuring Guards

Guards are used to protect state transitions. Interface *Guard* is used to do an evaluation where method has access to a *StateContext*.

```
@Configuration
@EnableStateMachine
public static class Config4 extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source(States.S1).target(States.S2)
                .event(Events.E1)
                .guard(guard())
                .and()
            .withExternal()
                .source(States.S2).target(States.S3)
                .event(Events.E2)
                .guardExpression("true");

    }

    @Bean
    public Guard<States, Events> guard() {
        return new Guard<States, Events>() {

            @Override
            public boolean evaluate(StateContext<States, Events> context) {
                return true;
            }
        };
    }

}
```

In above two different types of guard configurations are used. Firstly a simple *Guard* is created as a bean and attached to transition between states `S1` and `S2`.

Secondly a simple spel expression can be used as a guard where expression must return a `BOOLEAN` value. Behind a scenes this spel based guard is a *SpelExpressionGuard*. This was attached to transition between states `S2` and `S3`. Both guard in above sample always evaluate to true.

4.6 Configuring Actions

Actions can be defined with various steps within a state transitions.

```
@Configuration
@EnableStateMachine
public static class Config5 extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source(States.S1)
                .target(States.S2)
                .event(Events.E1)
                .action(action());
    }

    @Bean
    public Action<States, Events> action() {
        return new Action<States, Events>() {

            @Override
            public void execute(StateContext<States, Events> context) {
                // do something
            }
        };
    }

}
```

4.7 Configuring Pseudo States

Pseudo state configuration is usually done by configuring states and transitions. Pseudo states are automatically added to state machine as states.

Initial State

Simply mark a particular state as initial state by using `initial()` method. There are two methods where one takes extra argument to define an initial action. This initial action is good for example initialize extended state variables.

```
@Configuration
@EnableStateMachine
public static class Config11 extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1, initialAction())
                .end(States.SF)
                .states(EnumSet.allOf(States.class));
    }

    @Bean
    public Action<States, Events> initialAction() {
        return new Action<States, Events>() {

            @Override
            public void execute(StateContext<States, Events> context) {
                // do something initially
            }
        };
    }

}
```

Terminate State

Simply mark a particular state as end state by using `end()` method. This can be done max one time per individual sub-machine or region.

```
@Configuration
@EnableStateMachine
public static class Config1 extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1)
                .end(States.SF)
                .states(EnumSet.allOf(States.class));
    }

}
```

History State

History state can be defined once for each individual state machine. You need to choose its state identifier and `History.SHALLOW` or `History.DEEP` respectively.

```
@Configuration
@EnableStateMachine
public static class Config12 extends EnumStateMachineConfigurerAdapter<States3, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States3, Events> states)
            throws Exception {
        states
        .withStates()
            .initial(States3.S1)
            .state(States3.S2)
            .and()
            .withStates()
                .parent(States3.S2)
                .initial(States3.S2I)
                .state(States3.S21)
                .state(States3.S22)
                .history(States3.SH, History.SHALLOW);
    }

}
```

Choice State

Choice needs to be defined in both states and transitions to work properly. Mark particular state as choice state by using `choice()` method. This state needs to match source state when transition is configured for this choice.

Transition is configured using `withChoice()` where you define source state and `first/then/last` structure which is equivalent to normal `if/elseif/else`. With `first` and `then` you can specify a guard just like you'd use a condition with `if/elseif` clauses.

Transition needs to be able to exist so make sure `last` is used. Otherwise configuration is ill-formed.

```
@Configuration
@EnableStateMachine
public static class Config13 extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.SI)
                .choice(States.S1)
                .end(States.SF)
                .states(EnumSet.allOf(States.class));
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withChoice()
                .source(States.S1)
                .first(States.S2, s2Guard())
                .then(States.S3, s3Guard())
                .last(States.S4);
    }

    @Bean
    public Guard<States, Events> s2Guard() {
        return new Guard<States, Events>() {

            @Override
            public boolean evaluate(StateContext<States, Events> context) {
                return false;
            }
        };
    }

    @Bean
    public Guard<States, Events> s3Guard() {
        return new Guard<States, Events>() {

            @Override
            public boolean evaluate(StateContext<States, Events> context) {
                return true;
            }
        };
    }

}
```

Fork State

Fork needs to be defined in both states and transitions to work properly. Mark particular state as choice state by using `fork()` method. This state needs to match source state when transition is configured for this fork.

Target state needs to be a super state or immediate states in regions. Using a super state as target will take all regions into initial states. Targeting individual state give more controlled entry into regions.

```
@Configuration
@EnableStateMachine
public static class Config14 extends EnumStateMachineConfigurerAdapter<States2, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States2, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States2.S1)
                .fork(States2.S2)
                .state(States2.S3)
                .and()
                .withStates()
                    .parent(States2.S3)
                    .initial(States2.S2I)
                    .state(States2.S21)
                    .state(States2.S22)
                    .end(States2.S2F)
                    .and()
                .withStates()
                    .parent(States2.S3)
                    .initial(States2.S3I)
                    .state(States2.S31)
                    .state(States2.S32)
                    .end(States2.S3F);
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States2, Events> transitions)
            throws Exception {
        transitions
            .withFork()
                .source(States2.S2)
                .target(States2.S22)
                .target(States2.S32);
    }

}
```

Join State

Join needs to be defined in both states and transitions to work properly. Mark particular state as choice state by using `join()` method. This state doesn't need to match either source states or target state in a transition configuration.

Select one target state where transition goes when all source states has been joined. If you use state hosting regions as source, end states of a regions are used as joins. Otherwise you can pick any states from a regions.

```
@Configuration
@EnableStateMachine
public static class Config15 extends EnumStateMachineConfigurerAdapter<States2, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States2, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States2.S1)
                .state(States2.S3)
                .join(States2.S4)
                .and()
                .withStates()
                    .parent(States2.S3)
                    .initial(States2.S2I)
                    .state(States2.S21)
                    .state(States2.S22)
                    .end(States2.S2F)
                    .and()
                .withStates()
                    .parent(States2.S3)
                    .initial(States2.S3I)
                    .state(States2.S31)
                    .state(States2.S32)
                    .end(States2.S3F);
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States2, Events> transitions)
            throws Exception {
        transitions
            .withJoin()
                .source(States2.S2F)
                .source(States2.S3F)
                .target(States2.S5);
    }

}
```

5. State Machine Factories

There are use cases when state machine needs to be created dynamically instead of defining static configuration at compile time. For example if there are custom components which are using its own state machines and these components are created dynamically it is impossible to have a static state machined build during the application start. Internally state machines are always build via a factory interfaces and this then gives user an option to use this feature programmatically. Configuration for state machine factory is exactly same as you've seen in various examples in this document where state machine configuration is hard coded.

Actually creating a state machine using *@EnableStateMachine* will work via factory so *@EnableStateMachineFactory* is merely exposing that factory via its interface.

```
@Configuration
@EnableStateMachineFactory
public static class Config6
    extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1)
                .end(States.SF)
                .states(EnumSet.allOf(States.class));
    }

}
```

Now that you've used *@EnableStateMachineFactory* to create a factory instead of a state machine bean, it can be injected and used as is to request new state machines.

```
static class Bean3 {

    @Autowired
    StateMachineFactory<States, Events> factory;

    void method() {
        StateMachine<States,Events> stateMachine = factory.getStateMachine();
        stateMachine.start();
    }
}
```

5.1 Factory Limitations

Current limitation of factory is that all actions and guard it is associating with created state machine will share a same instances. This means that from your actions and guard you will need to specifically handle a case that same bean will be called by a different state machines. This limitation is something which will be resolved in future releases.

6. Using Actions

Actions are one of the most useful components from user perspective to interact and collaborate with a state machine. Actions can be executed in various places in a state machine and its states lifecycle like entering or exiting states or during a transitions.

```
@Override
public void configure(StateMachineStateConfigurer<States, Events> states)
        throws Exception {
    states
        .withStates()
            .initial(States.SI)
            .state(States.S1, action1(), action2())
            .state(States.S2, action1(), action2())
            .state(States.S3, action1(), action3());
}
```

Above `action1` and `action2` beans are attached to states entry and exit respectively.

```
@Bean
public Action<States, Events> action1() {
    return new Action<States, Events>() {

        @Override
        public void execute(StateContext<States, Events> context) {
        }
    };
}

@Bean
public BaseAction action2() {
    return new BaseAction();
}

@Bean
public SpelAction action3() {
    ExpressionParser parser = new SpelExpressionParser();
    return new SpelAction(
            parser.parseExpression(
                    "stateMachine.sendEvent(T(org.springframework.statemachine.docs.Events).E1)"));
}

static class BaseAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
    }
}

static class SpelAction extends SpelExpressionAction<States, Events> {

    public SpelAction(Expression expression) {
        super(expression);
    }
}
```

You can directly implement *Action* as an anonymous function or create a your own implementation and define appropriate implementation as a bean.

In `action3` a SpEL expression is used to send event **Events.E1** into a state machine.

> **Note**
>
> *StateContext* is described in section Chapter 8, *Using StateContext*.

6.1 SpEL Expressions with Actions

It is also possible to use SpEL expressions as a replacement for a full *Action* implementation.

7. Using Guards

Above `guard1` and `guard2` beans are attached to states entry and exit respectively.

```
@Override
public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
        throws Exception {
    transitions
        .withExternal()
            .source(States.SI).target(States.S1)
            .event(Events.E1)
            .guard(guard1())
            .and()
        .withExternal()
            .source(States.S1).target(States.S2)
            .event(Events.E1)
            .guard(guard2())
            .and()
        .withExternal()
            .source(States.S2).target(States.S3)
            .event(Events.E2)
            .guardExpression("extendedState.variables.get('myvar')");
}
```

You can directly implement *Guard* as an anonymous function or create a your own implementation and define appropriate implementation as a bean. In above sample `guardExpression` is simply checking if extended state variable `myvar` evaluates to *TRUE*.

```
@Bean
public Guard<States, Events> guard1() {
    return new Guard<States, Events>() {

        @Override
        public boolean evaluate(StateContext<States, Events> context) {
            return true;
        }
    };
}

@Bean
public BaseGuard guard2() {
    return new BaseGuard();
}

static class BaseGuard implements Guard<States, Events> {

    @Override
    public boolean evaluate(StateContext<States, Events> context) {
        return false;
    }
}
```

> **Note**
>
> *StateContext* is described in section Chapter 8, *Using StateContext*.

7.1 SpEL Expressions with Guards

It is also possible to use SpEL expressions as a replacement for a full *Guard* implementation. Only requirement is that expression needs to return a **Boolean** value to satisfy *Guard* implementation. This is demonstrated with a *guardExpression()* function which takes an expression as an argument.

8. Using StateContext

StateContext is a domain object representing a current status of a state machine within a transition or an action. Context gives an access to a various information like event, message headers, extended state variables, current transition and a top-level state machine in case there is a need to send events to a further processing.

9. Triggering Transitions

Driving a statemachine is done via transitions which are triggered by triggers. Currently supported triggers are *EventTrigger* and *TimerTrigger*.

9.1 EventTrigger

EventTrigger is the most useful trigger because it allows user to directly interact with a state machine by sending events to it. These events are also called signals. Trigger is added to a transition simply by associating a state to it during a configuration.

```
@Autowired
StateMachine<States, Events> stateMachine;

void signalMachine() {
    stateMachine.sendEvent(Events.E1);

    Message<Events> message = MessageBuilder
            .withPayload(Events.E2)
            .setHeader("foo", "bar")
            .build();
    stateMachine.sendEvent(message);
}
```

In above example we send an event using two different ways. Firstly we simply sent a type safe event using state machine api method `sendEvent(E event)`. Secondly we send event wrapped in a Spring messaging *Message* using api method `sendEvent(Message<E> message)` with a custom event headers. This allows user to add arbitrary extra information with an event which is then visible to *StateContext* when for example user is implementing actions.

9.2 TimerTrigger

TimerTrigger is useful when something needs to be triggered automatically without any user interaction. Trigger is added to a transition by associating a timer to it during a configuration.

10. Listening State Machine Events

There are use cases where you just want to know what is happening with a state machine, react to something or simply get logging for debugging purposes. SSM provides interfaces for adding listeners which then gives an option to get callback when various state changes, actions, etc are happening.

You basically have two options, either to listen Spring application context events or directly attach listener to a state machine. Both of these basically will provide same information where one is producing events as event classes and other producing callbacks via a listener interface. Both of these have pros and cons which will be discussed later.

10.1 Application Context Events

Application context events classes are *OnTransitionStartEvent*, *OnTransitionEvent*, *OnTransitionEndEvent*, *OnStateExitEvent*, *OnStateEntryEvent*, *OnStateChangedEvent*, *OnStateMachineStart* and *OnStateMachineStop*. These can be used as is with spring typed *ApplicationListener* class but they also share a common class *StateMachineEvent* which can be used to get statemachine related events.

```
static class StateMachineApplicationEventListener
    implements ApplicationListener<StateMachineEvent> {

    @Override
    public void onApplicationEvent(StateMachineEvent event) {
    }
}
```

10.2 State Machine Listener

Using *StateMachineListener* you can either extend it and implement all callback methods or use *StateMachineListenerAdapter* class which contains stub method implementations and choose which ones to override.

```
static class StateMachineEventListener
    extends StateMachineListenerAdapter<States, Events> {

    @Override
    public void stateChanged(State<States, Events> from, State<States, Events> to) {
    }

    @Override
    public void stateEntered(State<States, Events> state) {
    }

    @Override
    public void stateExited(State<States, Events> state) {
    }

    @Override
    public void transition(Transition<States, Events> transition) {
    }

    @Override
    public void transitionStarted(Transition<States, Events> transition) {
    }

    @Override
    public void transitionEnded(Transition<States, Events> transition) {
    }

    @Override
    public void stateMachineStarted(StateMachine<States, Events> stateMachine) {
    }

    @Override
    public void stateMachineStopped(StateMachine<States, Events> stateMachine) {
    }
}
```

In above example we simply created our own listener class *StateMachineEventListener* which extends *StateMachineListenerAdapter*.

Once you have your own listener defined, it can be registered into a state machine via its interface as shown below. It's just a matter of flavour if it's hooked up within a spring configuration or done manually at any time of application life-cycle.

```
static class Config7 {

    @Autowired
    StateMachine<States, Events> stateMachine;

    @Bean
    public StateMachineEventListener stateMachineEventListener() {
        StateMachineEventListener listener = new StateMachineEventListener();
        stateMachine.addStateListener(listener);
        return listener;
    }

}
```

10.3 Limitations and Problems

Spring application context is not a fastest event bus out there so it is advised to give some thought what is a rate of events state machine is sending. For better performance it may be better to use *StateMachineListener* interface. For this specific reason it is possible to use `contextEvents` flag with *@EnableStateMachine* and *@EnableStateMachineFactory* to disable Spring application context events as shown above.

```
@Configuration
@EnableStateMachine(contextEvents = false)
public static class Config8
    extends EnumStateMachineConfigurerAdapter<States, Events> {
}

@Configuration
@EnableStateMachineFactory(contextEvents = false)
public static class Config9
    extends EnumStateMachineConfigurerAdapter<States, Events> {
}
```

11. Context Integration

It is a little limited to do interaction with a state machine by either listening its events or using actions with states and transitions. Time to time this approach would be too limited and verbose to create interaction with the application a state machine is working with. For this specific use case we have made a spring style context integration which easily attach state machine functionality into your beans.

11.1 Annotation Support

@WithStateMachine annotation can be used to associate a state machine with a existing bean. Withing this annotation a propertys *source* and *target* can be used to qualify a transition

```
@WithStateMachine
static class Bean1 {

    @OnTransition(source = "S1", target = "S2")
    public void fromS1ToS2() {
    }
}
```

Default *@OnTransition* annotation can't be used with a state and event enums user have created due to java language limitations, thus string representation have to be used.

However if you want to have a type safe annotation it is possible to create a new annotation and use *@OnTransition* as meta annotation. This user level annotation can make a reference to actual states and events enums and framework will try to match these in a same way.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@OnTransition
static @interface StatesOnTransition {

    States[] source() default {};

    States[] target() default {};
}
```

Above we created a *@StatesOnTransition* annotation which defines `source` and `target` as a type safe manner.

```
@WithStateMachine
static class Bean2 {

    @StatesOnTransition(source = States.S1, target = States.S2)
    public void fromS1ToS2() {
    }
}
```

In your own bean you can then use this *@StatesOnTransition* as is and use type safe `source` and `target`.

Part III. State Machine Examples

This part of the reference documentation explains the use of state machines together with a sample code and a uml state charts. We do few shortcuts when representing relationship between a state chart, SSM configuration and what an application does with a state machine. For complete examples go and study the samples repository.

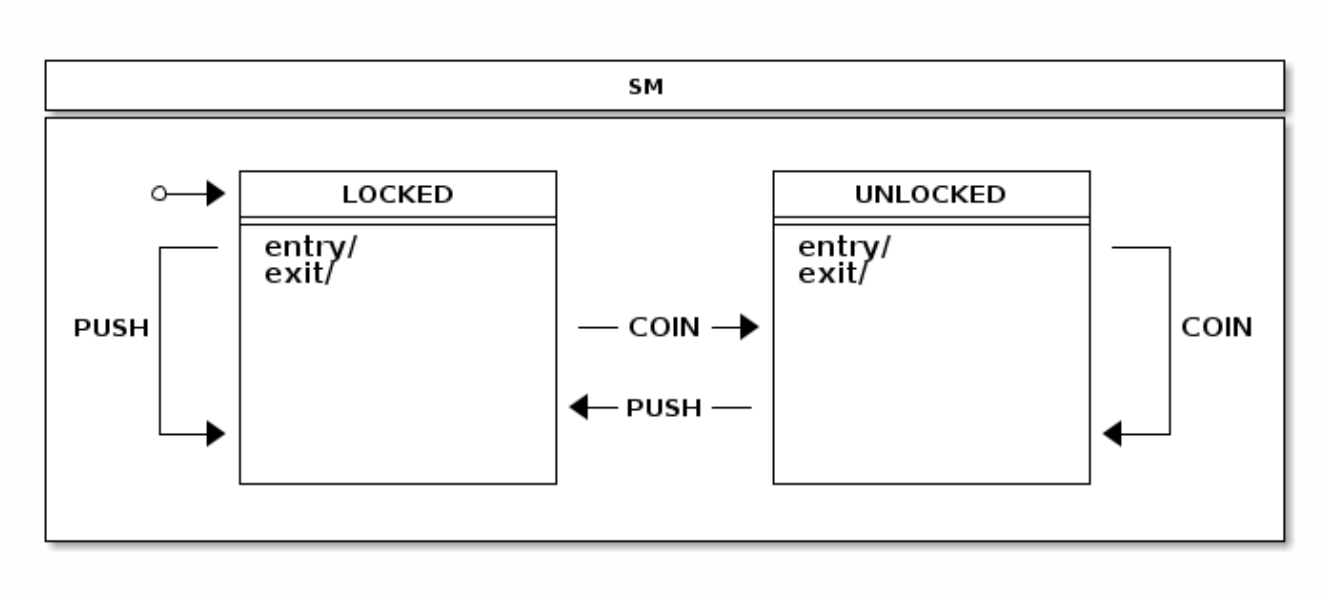
Samples are build directly from a main source distribution during a normal build cycle.

```
./gradlew clean build -x test
```

Every sample is located in its own directory under `spring-statemachine-samples`. Samples are based on spring-boot and spring-shell and you will find usual boot fat jars under every sample projects `build/libs` directory.

12. Turnstile

Turnstile is a simple device which gives you an access if payment is made and is a very simple to model using a state machine. In its simplest form there are only two states, `LOCKED` and `UNLOCKED`. Two events, `COIN` and `PUSH` can happen if you try to go through it or you make a payment.



States.

```
public static enum States {
    LOCKED, UNLOCKED
}
```

Events.

```
public static enum Events {
    COIN, PUSH
}
```

Configuration.

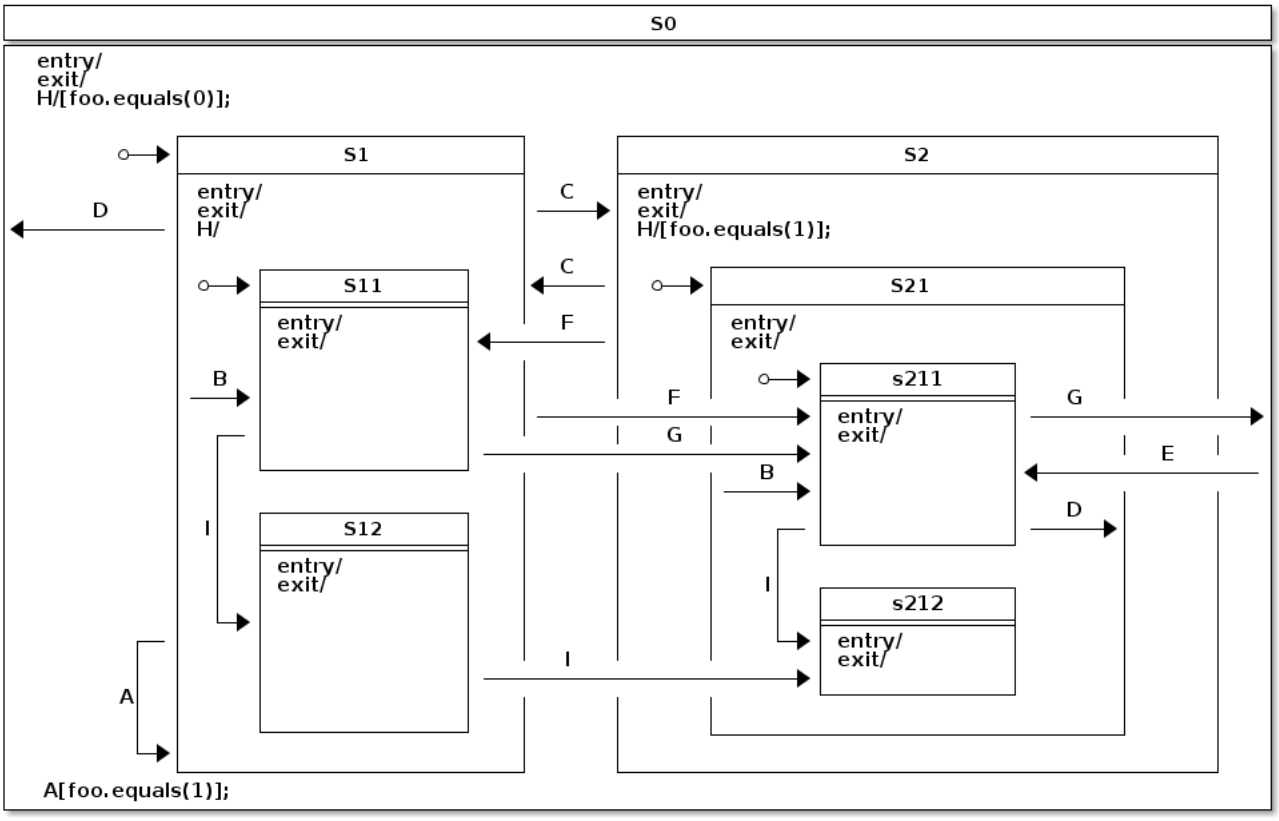
```
@Configuration
@EnableStateMachine
static class StateMachineConfig
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.LOCKED)
                .states(EnumSet.allOf(States.class));
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source(States.LOCKED)
                .target(States.UNLOCKED)
                .event(Events.COIN)
                .and()
            .withExternal()
                .source(States.UNLOCKED)
                .target(States.LOCKED)
                .event(Events.PUSH);
    }

}
```

You can see how this sample state machine interacts with event by running `turnstile` sample.

```
$ java -jar spring-statemachine-samples-turnstile-1.0.0.BUILD-SNAPSHOT.jar

sm>sm print
+---------------------------------------------------------------+
|                             SM                                |
+---------------------------------------------------------------+
|                                                               |
|         +----------------+          +----------------+         |
|     *-->|     LOCKED     |          |    UNLOCKED    |         |
|         +----------------+          +----------------+         |
|     +---| entry/         |          | entry/         |---+     |
|     |   | exit/          |          | exit/          |   |     |
|     |   |                |          |                |   |     |
| PUSH|   |                |---COIN-->|                |   |COIN |
|     |   |                |          |                |   |     |
|     |   |                |          |                |   |     |
|     |   |                |<--PUSH---|                |   |     |
|     +-->|                |          |                |<--+     |
|         |                |          |                |         |
|         +----------------+          +----------------+         |
|                                                               |
+---------------------------------------------------------------+

sm>sm start
State changed to LOCKED
State machine started

sm>sm event COIN
State changed to UNLOCKED
Event COIN send

sm>sm event PUSH
State changed to LOCKED
Event PUSH send
```

13. Showcase

Showcase is a complex state machine showing all possible transition topologies up to four levels of state nesting.



States.

```
public static enum States {
    S0, S1, S11, S12, S2, S21, S211, S212
}
```

Events.

```
public static enum Events {
    A, B, C, D, E, F, G, H, I
}
```

Configuration - states.

```
@Override
public void configure(StateMachineStateConfigurer<States, Events> states)
        throws Exception {
    states
        .withStates()
            .initial(States.S0, fooAction())
            .state(States.S0)
            .and()
            .withStates()
                .parent(States.S0)
                .initial(States.S1)
                .state(States.S1)
                .and()
                .withStates()
                    .parent(States.S1)
                    .initial(States.S11)
                    .state(States.S11)
                    .state(States.S12)
                    .and()
            .withStates()
                .parent(States.S0)
                .state(States.S2)
                .and()
                .withStates()
                    .parent(States.S2)
                    .initial(States.S21)
                    .state(States.S21)
                    .and()
                    .withStates()
                        .parent(States.S21)
                        .initial(States.S211)
                        .state(States.S211)
                        .state(States.S212);
}
```

Configuration - transitions.

```
@Override
public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
        throws Exception {
    transitions
        .withExternal()
            .source(States.S1).target(States.S1).event(Events.A)
            .guard(foo1Guard())
            .and()
        .withExternal()
            .source(States.S1).target(States.S11).event(Events.B)
            .and()
        .withExternal()
            .source(States.S21).target(States.S211).event(Events.B)
            .and()
        .withExternal()
            .source(States.S1).target(States.S2).event(Events.C)
            .and()
        .withExternal()
            .source(States.S2).target(States.S1).event(Events.C)
            .and()
        .withExternal()
            .source(States.S1).target(States.S0).event(Events.D)
            .and()
        .withExternal()
            .source(States.S211).target(States.S21).event(Events.D)
            .and()
        .withExternal()
            .source(States.S0).target(States.S211).event(Events.E)
            .and()
        .withExternal()
            .source(States.S1).target(States.S211).event(Events.F)
            .and()
        .withExternal()
            .source(States.S2).target(States.S11).event(Events.F)
            .and()
        .withExternal()
            .source(States.S11).target(States.S211).event(Events.G)
            .and()
        .withExternal()
            .source(States.S211).target(States.S0).event(Events.G)
            .and()
        .withInternal()
            .source(States.S0).event(Events.H)
            .guard(foo0Guard())
            .action(fooAction())
            .and()
        .withInternal()
            .source(States.S2).event(Events.H)
            .guard(foo1Guard())
            .action(fooAction())
            .and()
        .withInternal()
            .source(States.S1).event(Events.H)
            .and()
        .withExternal()
            .source(States.S11).target(States.S12).event(Events.I)
            .and()
        .withExternal()
            .source(States.S211).target(States.S212).event(Events.I)
            .and()
        .withExternal()
            .source(States.S12).target(States.S212).event(Events.I);

}
```

Configuration - actions and guard.

```
@Bean
public FooGuard foo0Guard() {
    return new FooGuard(0);
}

@Bean
public FooGuard foo1Guard() {
    return new FooGuard(1);
}

@Bean
public FooAction fooAction() {
    return new FooAction();
}
```

Action.

```
private static class FooAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        Map<Object, Object> variables = context.getExtendedState().getVariables();
        Integer foo = context.getExtendedState().get("foo", Integer.class);
        if (foo == null) {
            log.info("Init foo to 0");
            variables.put("foo", 0);
        } else if (foo == 0) {
            log.info("Switch foo to 1");
            variables.put("foo", 1);
        } else if (foo == 1) {
            log.info("Switch foo to 0");
            variables.put("foo", 0);
        }
    }
}
```

Guard.

```
private static class FooGuard implements Guard<States, Events> {

    private final int match;

    public FooGuard(int match) {
        this.match = match;
    }

    @Override
    public boolean evaluate(StateContext<States, Events> context) {
        Object foo = context.getExtendedState().getVariables().get("foo");
        return !(foo == null || !foo.equals(match));
    }
}
```

Lets go through what this state machine do when it's executed and we send various event to it.

```
sm>sm start
Entry state S0
Entry state S1
Entry state S11
Init foo to 0
State machine started

sm>sm event A
Event A send

sm>sm event C
Exit state S11
Exit state S1
Entry state S2
Entry state S21
Entry state S211
Event C send

sm>sm event H
Switch foo to 1
Event H send

sm>sm event C
Exit state S211
Exit state S21
Exit state S2
Entry state S1
Entry state S11
Event C send

sm>sm event A
Exit state S11
Exit state S1
Entry state S1
Entry state S11
Event A send
```

What happens in above sample:

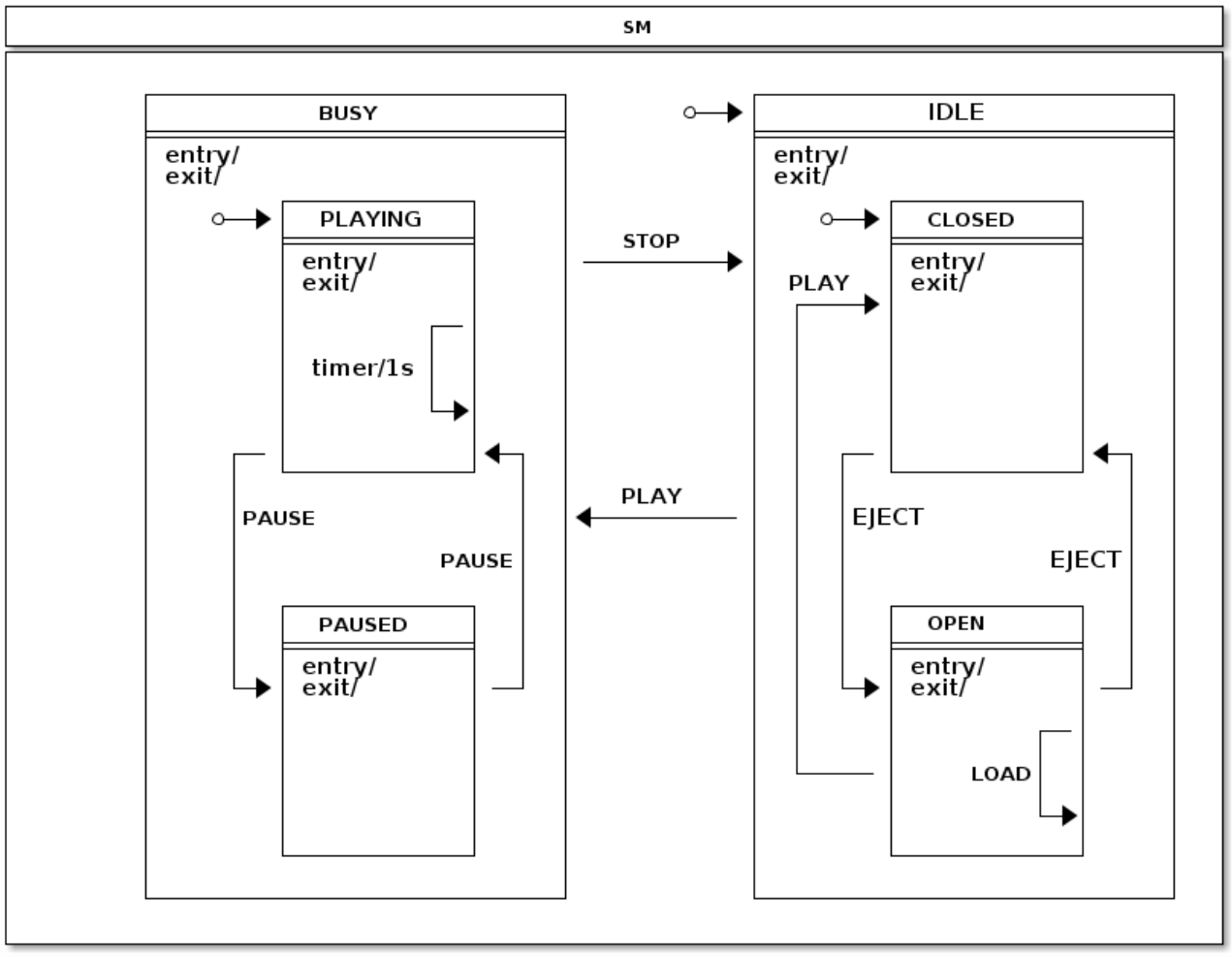
- State machine is started which takes it to its initial state *S11* via superstates *S1* and *S0*. Also extended state variable `foo` is init to `0`.
- We try to execute self transition in state *S1* with event *A* but nothing happens because transition is guarded by variable `foo` to be `1`.
- We send event *C* which takes us to other state machine where initial state *S211* and its superstates are entered. In there we can use event *H* which does a simple internal transition to flip variable `foo`. Then we simply go back using event *C*.
- Event *A* is sent again and now *S1* does a self transition because guard evaluates true.
- It's also worth to pay attention to how event *H* is handled in different states *S0*, *S1* and *S2*. This is a good example of how hierarchical states and their event handling works. If state *S2* is unable to handle event *H* due to guard condition, its parent is checked next. This guarantees that while on state *S2*, `foo` flag is always flipped around. However in state *S1* event *H* always match to its dummy transition without guard or action, not never happens.

14. CD Player

CD Player is a sample which resembles better use case of most of use have used in a real world. CD Player itself is a really simple entity where user can open a deck, insert or change a disk, then drive player functionality by pressing various buttons like *eject*, *play*, *stop*, *pause*, *rewind* and *backward*.

How many of use have really given a thought of what it will take to make a code for a CD Player which interacts with a hardware. Yes, concept of a player is overly simple but if you look behind a scenes things actually get a bit convoluted.

You've probably noticed that if your deck is open and you press play, deck will close and a song will start to play if CD was inserted in a first place. In a sense when deck is open you first need to close it and then try to start playing if cd is actually inserted. Hopefully you have now realised that a simple CD Player is not anymore so simple. Sure you can wrap all this with a simple class with few boolean variables and probably few nested if/else clauses, that will do the job, but what about if you need to make all this behaviour much more complex, do you really want to keep adding more flags and if/else clauses.



Lets go through how this sample and its state machine is designed and how those two interacts with each other. Below three config sections are used withing a *EnumStateMachineConfigurerAdapter*.

```
@Override
public void configure(StateMachineStateConfigurer<States, Events> states)
        throws Exception {
    states
        .withStates()
            .initial(States.IDLE)
            .state(States.IDLE)
            .and()
            .withStates()
                .parent(States.IDLE)
                .initial(States.CLOSED)
                .state(States.CLOSED, closedEntryAction(), null)
                .state(States.OPEN)
                .and()
        .withStates()
            .state(States.BUSY)
            .and()
            .withStates()
                .parent(States.BUSY)
                .initial(States.PLAYING)
                .state(States.PLAYING)
                .state(States.PAUSED);

}
```

```
@Override
public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
        throws Exception {
    transitions
        .withExternal()
            .source(States.CLOSED).target(States.OPEN).event(Events.EJECT)
            .and()
        .withExternal()
            .source(States.OPEN).target(States.CLOSED).event(Events.EJECT)
            .and()
        .withExternal()
            .source(States.OPEN).target(States.CLOSED).event(Events.PLAY)
            .and()
        .withExternal()
            .source(States.PLAYING).target(States.PAUSED).event(Events.PAUSE)
            .and()
        .withInternal()
            .source(States.PLAYING)
            .action(playingAction())
            .timer(1000)
            .and()
        .withInternal()
            .source(States.PLAYING).event(Events.BACK)
            .action(trackAction())
            .and()
        .withInternal()
            .source(States.PLAYING).event(Events.FORWARD)
            .action(trackAction())
            .and()
        .withExternal()
            .source(States.PAUSED).target(States.PLAYING).event(Events.PAUSE)
            .and()
        .withExternal()
            .source(States.BUSY).target(States.IDLE).event(Events.STOP)
            .and()
        .withExternal()
            .source(States.IDLE).target(States.BUSY).event(Events.PLAY)
            .action(playAction())
            .guard(playGuard())
            .and()
        .withInternal()
            .source(States.OPEN).event(Events.LOAD).action(loadAction());
}
```

```
@Bean
public ClosedEntryAction closedEntryAction() {
    return new ClosedEntryAction();
}

@Bean
public LoadAction loadAction() {
    return new LoadAction();
}

@Bean
public TrackAction trackAction() {
    return new TrackAction();
}

@Bean
public PlayAction playAction() {
    return new PlayAction();
}

@Bean
public PlayingAction playingAction() {
    return new PlayingAction();
}

@Bean
public PlayGuard playGuard() {
    return new PlayGuard();
}
```

What we did in above configuration:

- We used EnumStateMachineConfigurerAdapter to configure states and transitions.
- States *CLOSED* and *OPEN* are defined as substates of *IDLE*, states *PLAYING* and *PAUSED* are defined as substates of *BUSY*.
- With state *CLOSED* we added entry action as bean *closedEntryAction*.
- With transition we mostly mapped events to expected state transitions like *EJECT* closing and opening a deck, *PLAY*, *STOP* and *PAUSE* doing their natural transitions. Few words to mention what we did for other transitions.
  - With source state *PLAYING* we added a timer trigger which is needed to automatically track elapsed time within a playing track and to have facility to make a decision when to switch to next track.
  - With event *PLAY* if source state is *IDLE* and target state is *BUSY* we defined action *playAction* and guard *playGuard*.
  - With event *LOAD* and state *OPEN* we defined internal transition with action *loadAction* which will insert cd disc into extended state variables.
  - *PLAYING* state defined three internal transitions where one is triggered by a timer executing a *playingAction* which updates extended state variables. Other two transitions are with *trackAction* with different events, *BACK* and *FORWARD* respectively which handles when user wants to go back or forward in tracks.

This machine only have six states which are introduced as an enum.

```
public static enum States {
    // super state of PLAYING and PAUSED
    BUSY,
    PLAYING,
    PAUSED,
    // super state of CLOSED and OPEN
    IDLE,
    CLOSED,
    OPEN
}
```

Events represent, in a sense in this example, what buttons user would press and if user loads a cd disc into a deck.

```
public static enum Events {
    PLAY, STOP, PAUSE, EJECT, LOAD, FORWARD, BACK
}
```

Beans *cdPlayer* and *library* are just used with a sample to drive the application.

```
@Bean
public CdPlayer cdPlayer() {
    return new CdPlayer();
}

@Bean
public Library library() {
    return Library.buildSampleLibrary();
}
```

We can define extended state variable key as simple enums.

```
public static enum Variables {
    CD, TRACK, ELAPSEDTIME
}

public static enum Headers {
    TRACKSHIFT
}
```

We wanted to make this samply type safe so we're defining our own annotation *@StatesOnTransition* which have a mandatory meta annotation *@OnTransition*.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@OnTransition
public static @interface StatesOnTransition {

    States[] source() default {};

    States[] target() default {};

}
```

ClosedEntryAction is a entry action for state *CLOSED* to simply send and *PLAY* event to a statemachine if cd disc is present.

```
public static class ClosedEntryAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        if (context.getTransition() != null
                && context.getEvent() == Events.PLAY
                && context.getTransition().getTarget().getId() == States.CLOSED
                && context.getExtendedState().getVariables().get(Variables.CD) != null) {
            context.getStateMachine().sendEvent(Events.PLAY);
        }
    }
}
```

LoadAction is simply updating extended state variable if event headers contained information about a cd disc to load.

```
public static class LoadAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        Object cd = context.getMessageHeader(Variables.CD);
        context.getExtendedState().getVariables().put(Variables.CD, cd);
    }
}
```

PlayAction is simply resetting player elapsed time which is kept as an extended state variable.

```
public static class PlayAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        context.getExtendedState().getVariables().put(Variables.ELAPSEDTIME, 0l);
        context.getExtendedState().getVariables().put(Variables.TRACK, 0);
    }
}
```

PlayGuard is used to guard transition from *IDLE* to *BUSY* with event *PLAY* if extended state variable *CD* doesn't indicate that cd disc has been loaded.

```
public static class PlayGuard implements Guard<States, Events> {

    @Override
    public boolean evaluate(StateContext<States, Events> context) {
        ExtendedState extendedState = context.getExtendedState();
        return extendedState.getVariables().get(Variables.CD) != null;
    }
}
```

PlayingAction is updating extended state variable *ELAPSEDTIME* which cd player itself can read and update lcd status. Action also handles track shift if user is going back or forward in tracks.

```
public static class PlayingAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        Map<Object, Object> variables = context.getExtendedState().getVariables();
        Object elapsed = variables.get(Variables.ELAPSEDTIME);
        Object cd = variables.get(Variables.CD);
        Object track = variables.get(Variables.TRACK);
        if (elapsed instanceof Long) {
            long e = ((Long)elapsed) + 1000l;
            if (e > ((Cd) cd).getTracks()[((Integer) track)].getLength()*1000) {
                context.getStateMachine().sendEvent(MessageBuilder
                        .withPayload(Events.FORWARD)
                        .setHeader(Headers.TRACKSHIFT.toString(), 1).build());
            } else {
                variables.put(Variables.ELAPSEDTIME, e);
            }
        }
    }
}
```

TrackAction handles track shift action if user is going back or forward in tracks. If it is a last track of a cd, playing is stopped and *STOP* event sent to a state machine.

```
public static class TrackAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        Map<Object, Object> variables = context.getExtendedState().getVariables();
        Object trackshift = context.getMessageHeader(Headers.TRACKSHIFT.toString());
        Object track = variables.get(Variables.TRACK);
        Object cd = variables.get(Variables.CD);
        if (trackshift instanceof Integer && track instanceof Integer && cd instanceof Cd) {
            int next = ((Integer)track) + ((Integer)trackshift);
            if (next >= 0 &&  ((Cd)cd).getTracks().length > next) {
                variables.put(Variables.ELAPSEDTIME, 0l);
                variables.put(Variables.TRACK, next);
            } else if (((Cd)cd).getTracks().length <= next) {
                context.getStateMachine().sendEvent(Events.STOP);
            }
        }
    }
}
```

One other important aspect of a state machines is that they have their own responsibilies mostly around handling states and all application level logic should be kept outside. This means that application needs to have a ways to interact with a state machine and below sample is how cdplayer does it order to update lcd status. Also pay attention that we annotated *CdPlayer* with *@WithStateMachine* which instructs state machine to find methods from your pojo which are then called with various transitions.

```
@OnTransition(target = "BUSY")
public void busy(ExtendedState extendedState) {
    Object cd = extendedState.getVariables().get(Variables.CD);
    if (cd != null) {
        cdStatus = ((Cd)cd).getName();
    }
}
```

In above example we use *@OnTransition* annotation to hook a callback when transition happens with a target state *BUSY*.

```
@StatesOnTransition(target = {States.CLOSED, States.IDLE})
public void closed(ExtendedState extendedState) {
    Object cd = extendedState.getVariables().get(Variables.CD);
    if (cd != null) {
        cdStatus = ((Cd)cd).getName();
    } else {
        cdStatus = "No CD";
    }
    trackStatus = "";
}
```

@OnTransition we used above can only be used with strings which are matched from enums. *@StatesOnTransition* is then something what user can create into his own application to get a type safe annotation where a real enums can be used.

Lets see an example how this state machine actually works.

```
sm>sm start
Entry state IDLE
Entry state CLOSED
State machine started

sm>cd lcd
No CD

sm>cd library
0: Greatest Hits
  0: Bohemian Rhapsody  05:56
  1: Another One Bites the Dust  03:36
1: Greatest Hits II
  0: A Kind of Magic  04:22
  1: Under Pressure  04:08

sm>cd eject
Exit state CLOSED
Entry state OPEN

sm>cd load 0
Loading cd Greatest Hits

sm>cd play
Exit state OPEN
Entry state CLOSED
Exit state CLOSED
Exit state IDLE
Entry state BUSY
Entry state PLAYING

sm>cd lcd
Greatest Hits Bohemian Rhapsody 00:03

sm>cd forward

sm>cd lcd
Greatest Hits Another One Bites the Dust 00:04

sm>cd stop
Exit state PLAYING
Exit state BUSY
Entry state IDLE
Entry state CLOSED

sm>cd lcd
Greatest Hits
```

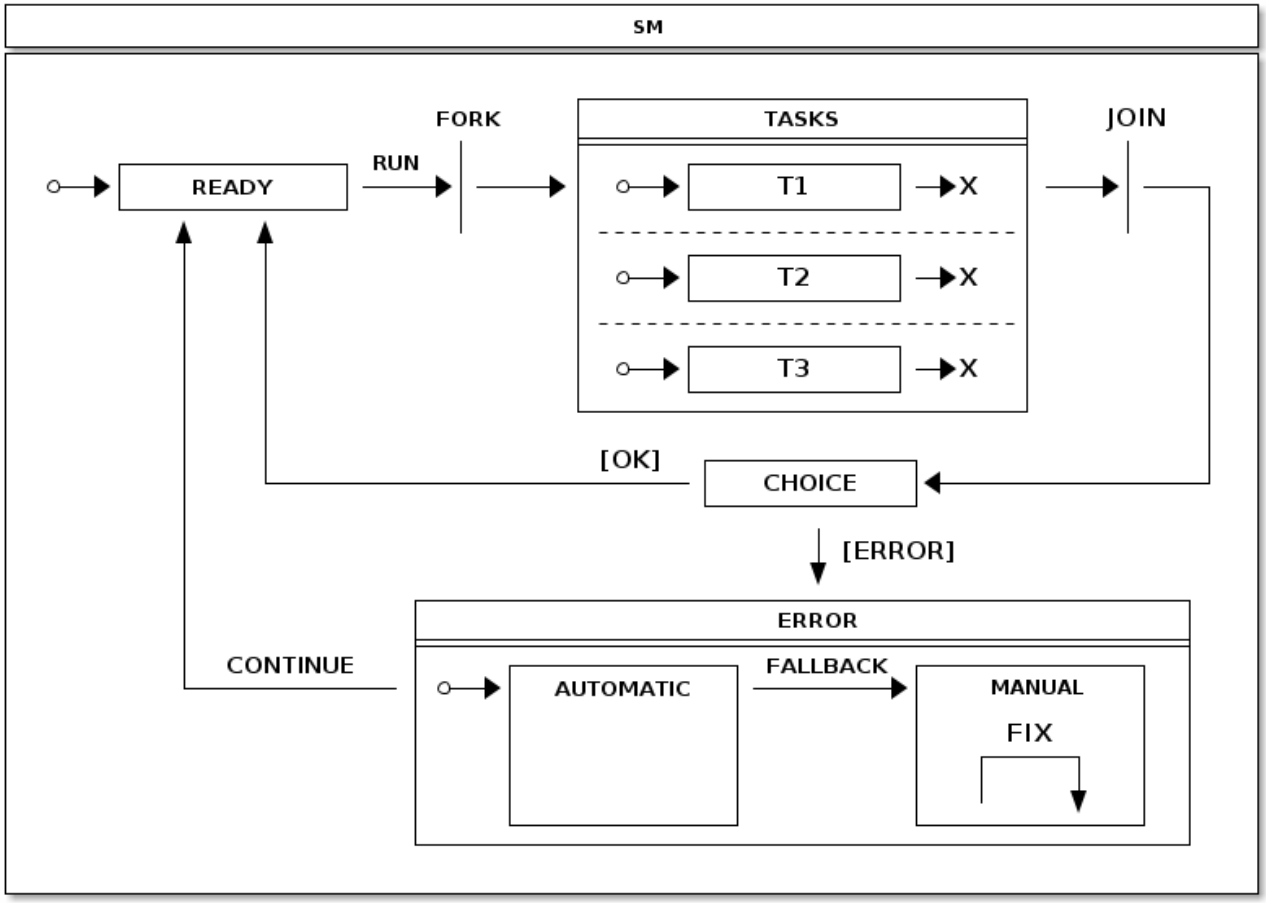
What happened in above run:

- State machine is started which causes machine to get initialized.

- CD Player lcd screen status is printed.
- CD Library is printed.
- CD Player deck is opened.
- CD with index 0 is loaded into a deck.
- Play is causing deck to get closed and immediate playing because cd was inserted.
- We print lcd status and request next track.
- We stop playing.

15. Tasks

Tasks is a sample demonstrating a parallel task handling within a regions and additionally adds an error handling to either automatically or manually fixing task problems before continuing back to a state where tasks can be run again.



On a high level what happens in this state machine is:

- We're always trying to get into READY state so that we can use event RUN to execute tasks.
- TASKS state which is composed with 3 independent regions has been put in a middle of FORK and JOIN states which will cause regions to go into its initial states and to be joined by end states.
- From JOIN state we go automatically into a CHOICE state which checks existence of error flags in extended state variables. Tasks can set these flags and it gives CHOICE state a possibility to go into ERROR state where errors can be handled either automatically or manually.
- AUTOMATIC state in ERROR can try to automatically fix error and goes back to READY if it succeed to do so. If error is something what can't be handled automatically, user intervention is needed and machine is put into MANUAL state via FALLBACK event.

States.

```
public static enum States {
    READY,
    FORK, JOIN, CHOICE,
    TASKS, T1, T1E, T2, T2E, T3, T3E,
    ERROR, AUTOMATIC, MANUAL
}
```

Events.

```
public static enum Events {
    RUN, FALLBACK, CONTINUE, FIX;
}
```

Configuration - states.

```
@Override
public void configure(StateMachineStateConfigurer<States, Events> states)
        throws Exception {
    states
        .withStates()
            .initial(States.READY)
            .fork(States.FORK)
            .state(States.TASKS)
            .join(States.JOIN)
            .choice(States.CHOICE)
            .state(States.ERROR)
            .and()
            .withStates()
                .parent(States.TASKS)
                .initial(States.T1)
                .end(States.T1E)
                .and()
            .withStates()
                .parent(States.TASKS)
                .initial(States.T2)
                .end(States.T2E)
                .and()
            .withStates()
                .parent(States.TASKS)
                .initial(States.T3)
                .end(States.T3E)
                .and()
            .withStates()
                .parent(States.ERROR)
                .initial(States.AUTOMATIC)
                .state(States.AUTOMATIC, automaticAction(), null)
                .state(States.MANUAL);
}
```

Configuration - transitions.

```
@Override
public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
        throws Exception {
    transitions
        .withExternal()
            .source(States.READY).target(States.FORK)
            .event(Events.RUN)
            .and()
        .withFork()
            .source(States.FORK).target(States.TASKS)
            .and()
        .withExternal()
            .source(States.T1).target(States.T1E)
            .and()
        .withExternal()
            .source(States.T2).target(States.T2E)
            .and()
        .withExternal()
            .source(States.T3).target(States.T3E)
            .and()
        .withJoin()
            .source(States.TASKS).target(States.JOIN)
            .and()
        .withExternal()
            .source(States.JOIN).target(States.CHOICE)
            .and()
        .withChoice()
            .source(States.CHOICE)
            .first(States.ERROR, tasksChoiceGuard())
            .last(States.READY)
            .and()
        .withExternal()
            .source(States.ERROR).target(States.READY)
            .event(Events.CONTINUE)
            .and()
        .withExternal()
            .source(States.AUTOMATIC).target(States.MANUAL)
            .event(Events.FALLBACK)
            .and()
        .withInternal()
            .source(States.MANUAL)
            .action(fixAction())
            .event(Events.FIX);
}
```

Guard below is guarding choice entry into a ERROR state and needs to return TRUE if error has happened. For this guard simply checks that all extended state variables(T1, T2 and T3) are TRUE.

```
@Bean
public Guard<States, Events> tasksChoiceGuard() {
    return new Guard<States, Events>() {

        @Override
        public boolean evaluate(StateContext<States, Events> context) {
            Map<Object, Object> variables = context.getExtendedState().getVariables();
            return !(ObjectUtils.nullSafeEquals(variables.get("T1"), true)
                    && ObjectUtils.nullSafeEquals(variables.get("T2"), true)
                    && ObjectUtils.nullSafeEquals(variables.get("T3"), true));
        }
    };
}
```

Actions below will simply send event to a state machine to request next step which would be either fallback or continue back to ready.

```
@Bean
public Action<States, Events> automaticAction() {
    return new Action<States, Events>() {

        @Override
        public void execute(StateContext<States, Events> context) {
            Map<Object, Object> variables = context.getExtendedState().getVariables();
            if (ObjectUtils.nullSafeEquals(variables.get("T1"), true)
                    && ObjectUtils.nullSafeEquals(variables.get("T2"), true)
                    && ObjectUtils.nullSafeEquals(variables.get("T3"), true)) {
                context.getStateMachine().sendEvent(Events.CONTINUE);
            } else {
                context.getStateMachine().sendEvent(Events.FALLBACK);
            }
        }
    };
}

@Bean
public Action<States, Events> fixAction() {
    return new Action<States, Events>() {

        @Override
        public void execute(StateContext<States, Events> context) {
            Map<Object, Object> variables = context.getExtendedState().getVariables();
            variables.put("T1", true);
            variables.put("T2", true);
            variables.put("T3", true);
            context.getStateMachine().sendEvent(Events.CONTINUE);
        }
    };
}
```

Currently default region execution is synchronous but it can be changed to asynchronous by changing `TaskExecutor`. Task will simulate work by sleeping 2 seconds so you'll able to see how actions in regions are executed parallel.

```
@Bean
public TaskExecutor taskExecutor() {
    ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
    taskExecutor.setCorePoolSize(5);
    return taskExecutor;
}
```

Lets see an examples how this state machine actually works.

```
sm>sm start
State machine started
Entry state READY

sm>tasks run
Entry state TASKS
run task on T3
run task on T2
run task on T1
run task on T2 done
run task on T1 done
run task on T3 done
Entry state T2
Entry state T3
Entry state T1
Entry state T1E
Entry state T2E
Entry state T3E
Exit state TASKS
Entry state JOIN
Exit state JOIN
Entry state READY
```

In above we can execute tasks multiple times.

```
sm>tasks list
Tasks {T1=true, T3=true, T2=true}

sm>tasks fail T1

sm>tasks list
Tasks {T1=false, T3=true, T2=true}

sm>tasks run
Entry state TASKS
run task on T1
run task on T3
run task on T2
run task on T1 done
run task on T3 done
run task on T2 done
Entry state T1
Entry state T3
Entry state T2
Entry state T1E
Entry state T2E
Entry state T3E
Exit state TASKS
Entry state JOIN
Exit state JOIN
Entry state ERROR
Entry state AUTOMATIC
Exit state AUTOMATIC
Exit state ERROR
Entry state READY
```

In above, if we simulate failure for task T1, it is fixed automatically.

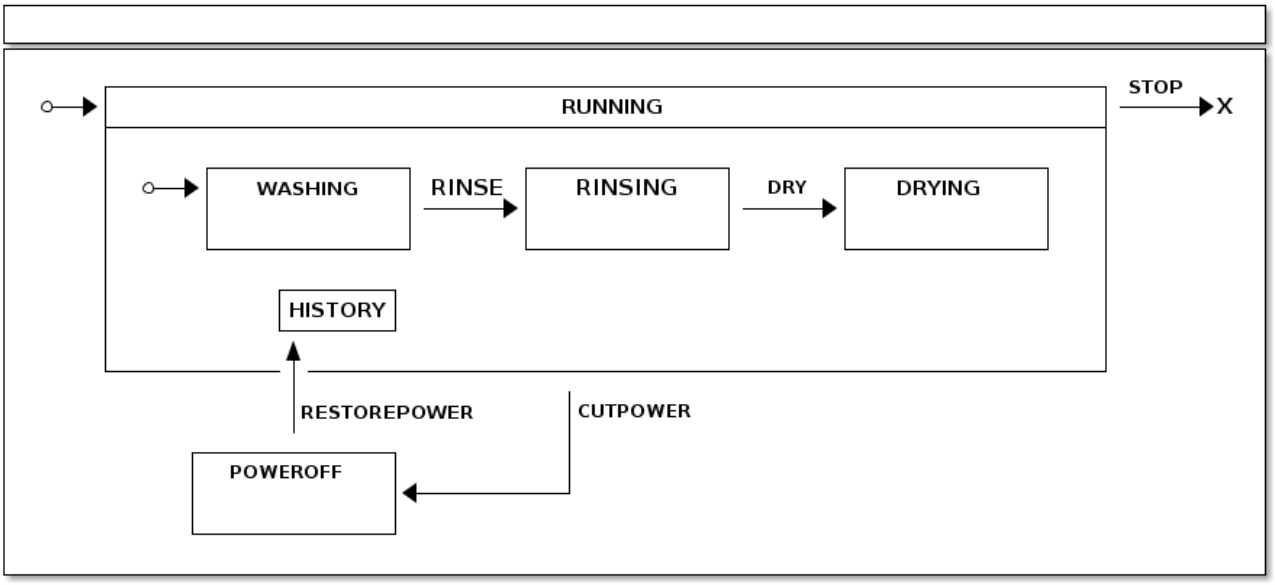
```
sm>tasks list
Tasks {T1=true, T3=true, T2=true}

sm>tasks fail T2

sm>tasks run
Entry state TASKS
run task on T2
run task on T1
run task on T3
run task on T2 done
run task on T1 done
run task on T3 done
Entry state T2
Entry state T1
Entry state T3
Entry state T1E
Entry state T2E
Entry state T3E
Exit state TASKS
Entry state JOIN
Exit state JOIN
Entry state ERROR
Entry state AUTOMATIC
Exit state AUTOMATIC
Entry state MANUAL

sm>tasks fix
Exit state MANUAL
Exit state ERROR
Entry state READY
```

In above if we simulate failure for either task T2 or T3, state machine goes to MANUAL state where problem needs to be fixed manually before we’re able to go back to READY state.

16. Washer

Washer is a sample demonstrating a use of a history state to recover a running state configuration with a simulated power off situation.

Anyone ever used a washing machine knows that if you can somehow pause the program it will continue from a same state when lid is closed. This kind of behaviour can be implemented in a state machine by using a history pseudo state.



States.

```
public static enum States {
    RUNNING, HISTORY, END,
    WASHING, RINSING, DRYING,
    POWEROFF
}
```

Events.

```
public static enum Events {
    RINSE, DRY, STOP,
    RESTOREPOWER, CUTPOWER
}
```

Configuration - states.

```
@Override
public void configure(StateMachineStateConfigurer<States, Events> states)
        throws Exception {
    states
        .withStates()
            .initial(States.RUNNING)
            .state(States.POWEROFF)
            .end(States.END)
            .and()
            .withStates()
                .parent(States.RUNNING)
                .initial(States.WASHING)
                .state(States.RINSING)
                .state(States.DRYING)
                .history(States.HISTORY, History.SHALLOW);
}
```

Configuration - transitions.

```
@Override
public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
        throws Exception {
    transitions
        .withExternal()
            .source(States.WASHING).target(States.RINSING)
            .event(Events.RINSE)
            .and()
        .withExternal()
            .source(States.RINSING).target(States.DRYING)
            .event(Events.DRY)
            .and()
        .withExternal()
            .source(States.RUNNING)    .target(States.POWEROFF)
            .event(Events.CUTPOWER)
            .and()
        .withExternal()
            .source(States.POWEROFF).target(States.HISTORY)
            .event(Events.RESTOREPOWER)
            .and()
        .withExternal()
            .source(States.RUNNING).target(States.END)
            .event(Events.STOP);
}
```

Lets see an example how this state machine actually works.

```
sm>sm start
Entry state RUNNING
Entry state WASHING
State machine started

sm>sm event RINSE
Exit state WASHING
Entry state RINSING
Event RINSE send

sm>sm event DRY
Exit state RINSING
Entry state DRYING
Event DRY send

sm>sm event CUTPOWER
Exit state DRYING
Exit state RUNNING
Entry state POWEROFF
Event CUTPOWER send

sm>sm event RESTOREPOWER
Exit state POWEROFF
Entry state RUNNING
Entry state WASHING
Entry state DRYING
Event RESTOREPOWER send
```

What happened in above run:

- State machine is started which causes machine to get initialized.
- We go to RINSING state.
- We go to DRYING state.
- We cut power and go to POWEROFF state.
- State is restored via HISTORY state which takes state machine back to its previous known state.

Part IV. FAQ

This chapter tries to give solutions to question user is most likely to ask.

17. State Changes

I want to transit to next state automatically.

There are few choices a state machine developer can choose.

- Implement an action and send appropriate event into a state machine which triggers a transition into a proper target state.
- Define deferred event within a state and before sending an event send a event which will be deferred and thus causing next appropriate state transition when it is more convenient to handle that event.
- Implement a triggerless transition which will automatically cause state transition into a next state when state has entry and its actions has been completed.

18. Extented State

How I can initialise variables on state machine start.

Important concept in a state machine is that nothing really happens unless there is a trigger which is causing a state transition which then can fire actions. However, having said that, Spring Statemachine always have an initial transition when state machine is started. With this initial transition user can execute a simple action which within a *StateContext* can do whatever it likes with an extended state variables.

Part V. Appendices

Appendix A. Support Content

This appendix provides generic information about used classes and material in this reference documentation.

A.1 Classes Used in This Document

```
public enum States {
    SI,S1,S2,S3,S4,SF
}
```

```
public enum States2 {
    S1,S2,S3,S4,S5,
    S2I,S21,S22,S2F,
    S3I,S31,S32,S3F
}
```

```
public enum States3 {
    S1,S2,SH,
    S2I,S21,S22,S2F
}
```

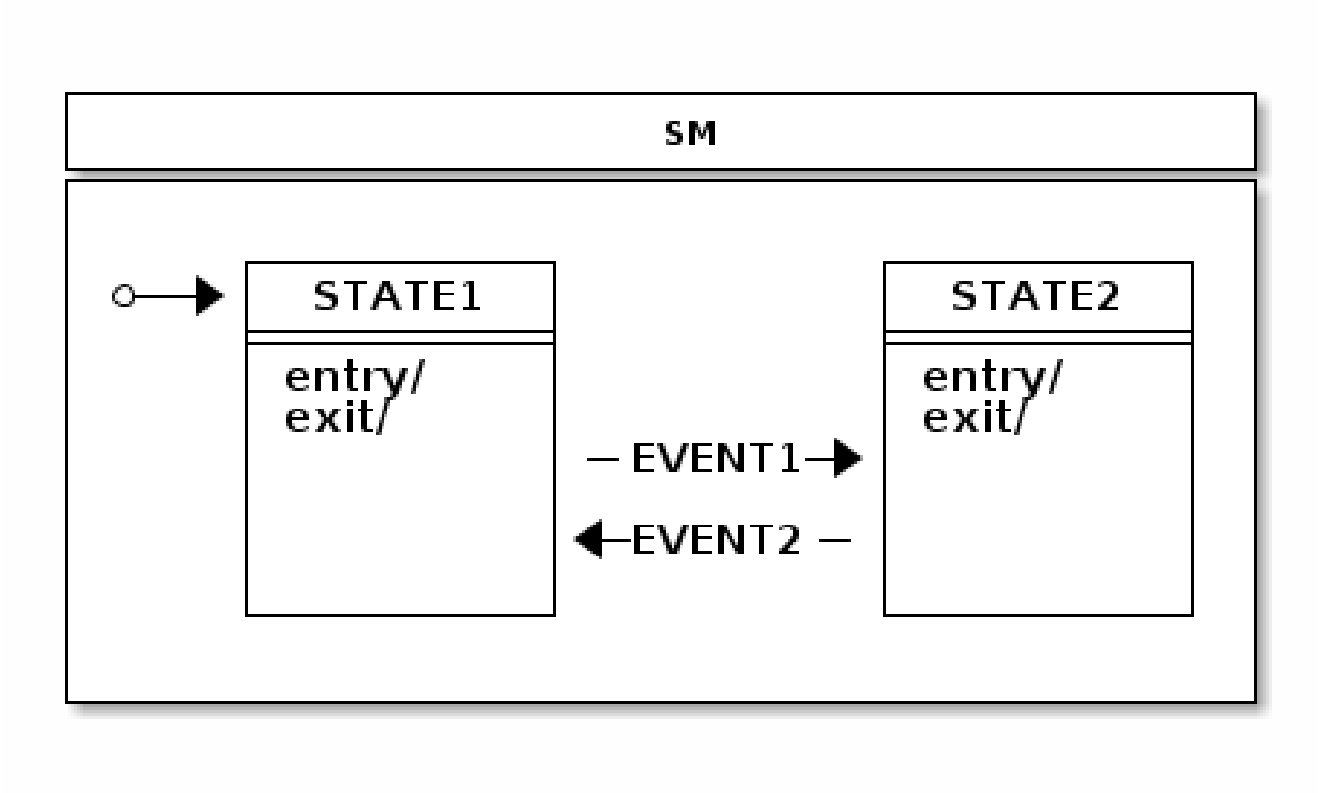
```
public enum Events {
    E1,E2,E3,E4,EF
}
```

Appendix B. State Machine Concepts

This appendix provides generic information about state machines.

B.1 Quick Example

Assuming we have states *STATE1*, *STATE2* and events *EVENT1*, *EVENT2*, logic of state machine can be defined as shown in below quick example.




```
static enum States {
    STATE1, STATE2
}

static enum Events {
    EVENT1, EVENT2
}
```

```
@Configuration
@EnableStateMachine
static class Config1 extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.STATE1)
                .states(EnumSet.allOf(States.class));
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source(States.STATE1).target(States.STATE2)
                .event(Events.EVENT1)
                .and()
            .withExternal()
                .source(States.STATE2).target(States.STATE1)
                .event(Events.EVENT2);
    }
}
```

```
@WithStateMachine
static class MyBean {

    @OnTransition(target = "STATE1")
    void toState1() {
    }

    @OnTransition(target = "STATE2")
    void toState2() {
    }
}
```

```
static class MyApp {

    @Autowired
    StateMachine<States, Events> stateMachine;

    void doSignals() {
        stateMachine.sendEvent(Events.EVENT1);
        stateMachine.sendEvent(Events.EVENT2);
    }
}
```

B.2 Glossary

State Machine

Main entity driving a collection of states together with regions, transitions and events.

State

A state models a situation during which some invariant condition holds. State is the main entity of a state machine where state changes are driven by an events.

Transition

A transition is a relationship between a source state and a target state. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type.

Event
An entity which is send to a state machine which then drives a various state changes.

Initial State
A special state in which the state machine starts. Initial state is always bound to a particular state machine or a region. A state machine with a multiple regions may have a multiple initial states.

End State
Also called as a final state is a special kind of state signifying that the enclosing region is completed. If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed, then it means that the entire state machine is completed.

History State
A pseudo state which allows a state machine to remember its last active state. Two types of history state exists, *shallow* which only remember top level state and *deep* which remembers active states in a sub-machines.

Choice State
A pseudo state which allows to make a transition choice based of i.e. event headers or extended state variables.

Fork State
A pseudo state which gives a controlled entry into a regions.

Join State
A pseudo state which gives a controlled exit from a regions.

Region
A region is an orthogonal part of either a composite state or a state machine. It contains states and transitions.

Guard
Is a boolean expression evaluated dynamically based on the value of extended state variables and event parameters. Guard conditions affect the behavior of a state machine by enabling actions or transitions only when they evaluate to TRUE and disabling them when they evaluate to FALSE.

Action
A action is a behaviour executed during the triggering of the transition.

B.3 A State Machines Crash Course

This appendix provides generic crash course to a state machine concepts.

States

A state is a model which a state machine can be in. It is always easier to describe state as a real world example rather than trying to abstract concepts with a generic documentation. For example lets take a simple example of a keyboard most of us are using every single day. If you have a full keyboard which has normal keys on a left side and the numeric keypad on a right side you may have noticed that the numeric keypad may be in a two different states depending whether numlock is activated or not. If it is not active then typing will result navigation using arrows, etc. If numpad is active then typing will result numbers to be used. Essentially numpad part of a keyboard can be in two different states.

To relate state concept to programming it means that instead of using flags, nested if/else/break clauses or other impractical logic you simply rely on state, state variables or other interaction with a state machine.

Pseudo States

PseudoState is a special type of state which usually introduces more higher level logic into a state machine by either giving a state a special meaning like initial state. State machine can then internally react to these states by doing various actions available in UML state machine concepts.

Initial

Initial pseudostate state is always needed for every single state machine whether you have a simple one level state machine or more complex state machine composed with submachines or regions. Initial state simple defines where state machine should go when it starts and without it state machine is ill-formed.

End

Terminate pseudostate which is also called as end state will indicate that a particular state machine has reached its final state. Effectively this mean that a state machine will no longer process any events and will not transit to any other state. However in a case of submachines are regions, state machine is able to restart from its terminal state.

Choice

Choice pseudostate is used to choose a dynamic conditional branch of a transition from this state. Dynamic condition is evaluated by guards so that at least one and at most one branch is selected. Usually a simple if/elseif/else structure is used to make sure that at least one branch is selected. Otherwise state machine might end up in a deadlock and configuration would be ill-formed.

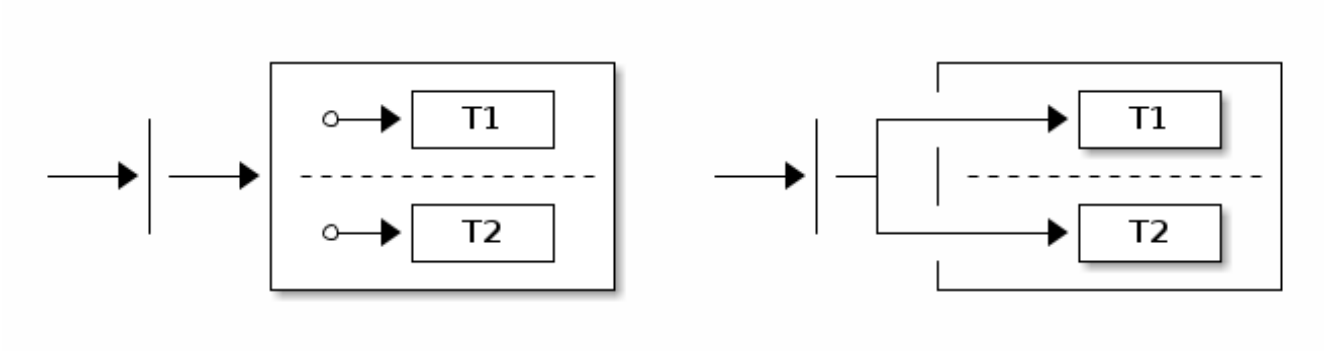
History

History pseudostate can be used to remember a last active state configuration. After state machine has been exited, history state can be used to restore previous knows configuration. There are two types of history states available, *SHALLOW* only remember active state of a state machine itself while *DEEP* also remembers nested states.

History state could be implemented externally by listening state machine events but this would soon make logic very difficult to work with, especially if state machine contains complex nested structures. Letting state machine itself to handle recording of history states makes things much simpler. What is left for user to do is simply do a transition into a history state and state machine will hand the needed logic to go back to its last known recorded state.

Fork

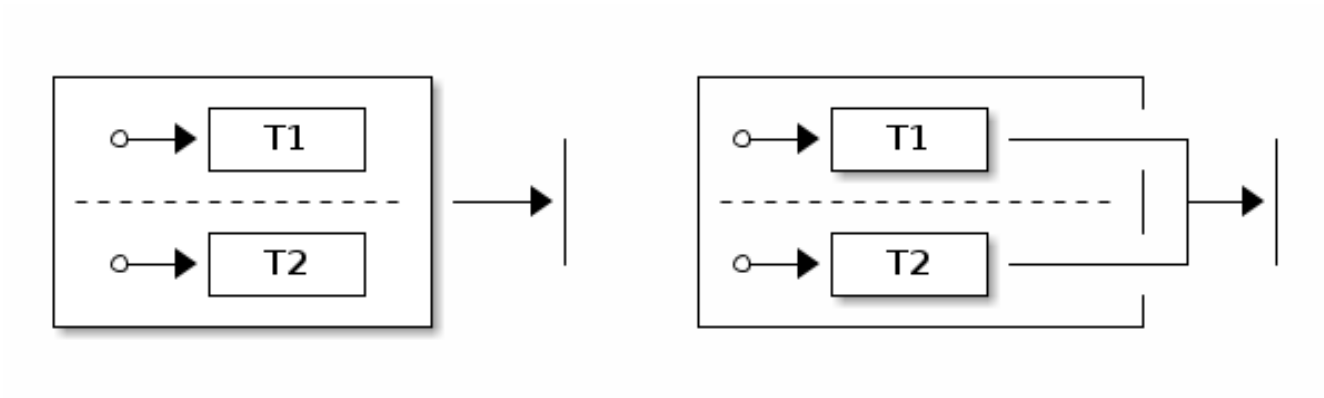
Fork pseudostate can be used to do an explicit entry into one or more regions.



Target state can be a parent state hosting regions, which simply means that regions are activated by entering its initial states. It's also possible to add targets directly to any state in a region which allows more controlled entry into a state.

Join

Join pseudostate is used to merge several transitions together originating from different regions. It it generally used to wait and block for participating regions to get into its join target states.



Source state can be a parent state hosting regions, which means that join states will be a terminate states of a participating regions. It's also possible to define source states to be any state in a regions which allows controlled exit from a regions.

Guard Conditions

Guard conditions are expressions which evaluates either to **TRUE** or **FALSE** based on extended state variables and event parameters. Guards are used with actions and transitions to dynamically choose if particular action or transition should be executed. Aspects of guards, event parameters and extended state variables are simply to make state machine design much more simple.

Events

Event is the most used trigger behaviour to drive a state machine. There are other ways to trigger behaviour to happen in state machine like a timer but events are the ones which really allows user to interact with a state machine. Events are also called as signals to possibly alter a state machine state.

Transitions

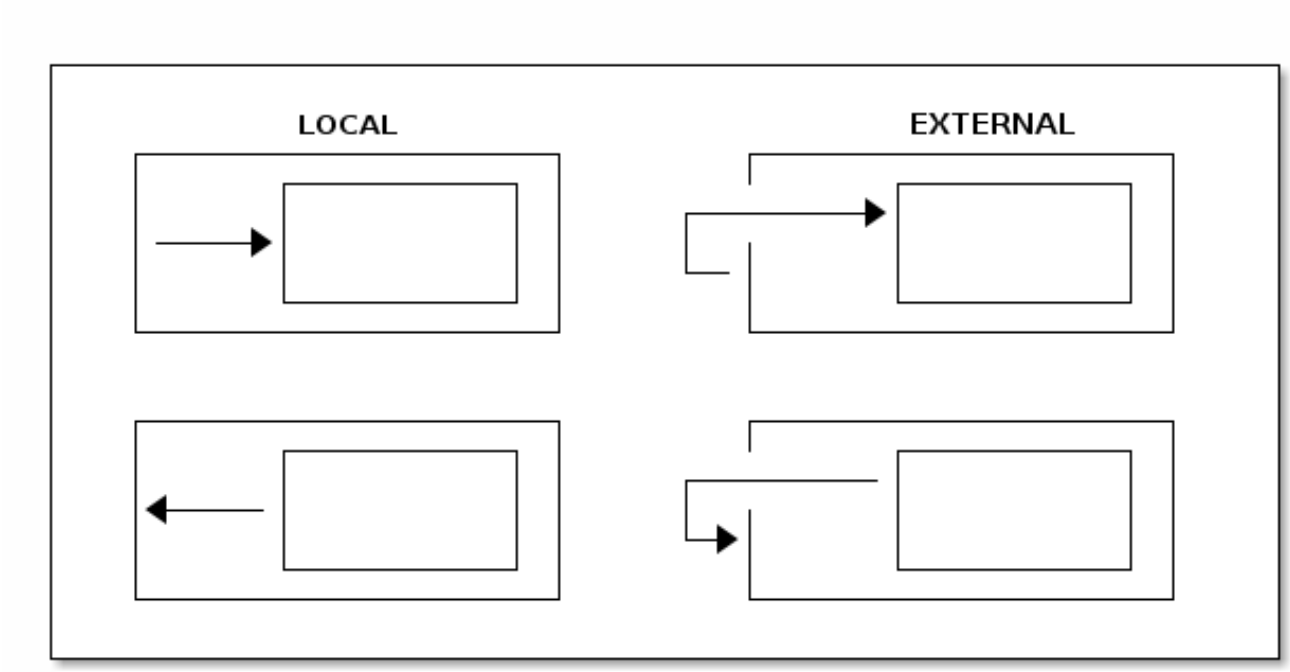
A transition is a relationship between a source state and a target state. A switch from a state to another is a *state transition* caused by a *trigger*.

Internal Transition

Internal transition is used when action needs to be executed without causing a state transition. With internal transition source and target state is always a same and it is identical with self-transition in the absence of state entry and exit actions.

External vs. Local Transition

Most of the cases external and local transition are functionally equivalent except in cases where transition is happening between super and sub states. Local transition doesn't cause exit and entry to source state if target state is a substate of a source state. Other way around, local transition doesn't cause exit and entry to target state if target is a superstate of a source state.



Above image shows a different between local and external transitions with a very simplistic super and sub states.

Actions

Actions are the ones which really glues state machine state changes with a user's own code. State machine can execute action on various changes and steps in a state machine like entering or exiting a state, or doing a state transition.

Actions usually have access to a state context which gives running code a choice to interact with a state machine in a various ways. State context i.e. is exposing a whole state machine so user can access extended state variables, event headers if transition is based on an event, or actual transition where it is possible to see more detailed where this state change is coming from and where it is going.

Hierarchical State Machines

Concept of a hierarchical state machine is used to simplify state design when particular states can only exist together.

Hierarchical states are really an innovation in UML state machine over a traditional state machines like Mealy or Moore machines. Hierarchical states allows to define some level of abstraction is a sense how java developer would define a class structure with abstract classes. For example having a nested state machine user is able to define transition on a multiple level of states possibly with a different conditions. State machine will always try to see if current state is able to handle an event together with a transition guard conditions. If these conditions are not evaluated to true, state machine will simply see what a super state can handle.

Regions

Regions which are also called as orthogonal regions are usually viewed as exclusive OR operation applied to a states. Concept of a region in terms of a state machine is usually a little difficult to understand but things gets a little simpler with a simple example.

Some of us have a full size keyboard with main keys on a left side and numeric keys on a right side. You've probably noticed that both sides really have their own state which you see if you press a numlock key which only alters behaviour of numbad itself. If you don't have a full size keyboard you can buy a simple external usb numbad having only numbad part of a keys. If left and right side can freely exist without the other they must have a totally different states which means they are operating on different state machines.

It would be a little inconvenient to handle two different statemachines as totally separate entities because in a sense they are still working together in a sense. This is why orthogonal regions can combine together a multiple simultaneous states within a single state in a state machine.