

C++ State Template Class Library STTCL Concept

Version 2.0 / Mar 25, 2012

Author: Günther Makulik (g-makulik@t-online.de)

Table of Contents

1	Overview.....	4
1.1	The GoF State pattern.....	4
1.2	Mapping UML State Machine notation elements.....	5
2	Implementation Design.....	14
2.1	Aspect Oriented Modelling of State Machines.....	15
2.1.1	Composite state aspect.....	15
2.1.2	Concurrency aspect.....	16
2.1.3	Direct transition aspect.....	16
2.2	Configuring the STTCL library.....	17
2.2.1	Configuring STTCL STL dependency.....	17
2.2.2	Configuring STTCL builtin concurrency implementations.....	17
2.2.3	Providing custom implementations for concurrency.....	18
2.3	STTCL Base Classes.....	19
2.3.1	sttcl::StateMachine<>.....	19
2.3.2	sttcl::State<>.....	21
2.3.3	sttcl::ActiveState<>.....	22
2.3.4	sttcl::CompositeState<>.....	25
2.3.5	sttcl::ConcurrentCompositeState<>.....	27
2.3.6	sttcl::Region<>.....	28
2.4	STTCL configuration adapters.....	31
2.4.1	sttcl::internal::SttclThread<>.....	32
2.4.2	sttcl::internal::SttclMutex<>.....	33
2.4.3	sttcl::internal::SttclSemaphore<>.....	33
2.4.4	sttcl::TimeDuration<>.....	34
2.4.5	sttcl::internal::SttclEventQueue<>.....	35
3	STTCL Demo applications.....	36
3.1	Demo1.....	36
3.2	Demo2.....	37
3.3	Demo3.....	38
3.4	Demo3a.....	40
3.5	Demo3b.....	41
3.6	Demo4.....	42
3.7	Demo5.....	43
3.8	Demo5a.....	43

History:

Version	Date	Changes	Author
2.0	02/11/12	Initial version	G. Makulik
2.0	02/16/12	Completed chapter 2.3 STTCL Base Classes Added state diagrams for Demo applications	G. Makulik

1 Overview

The C++ **State Template Class Library** provides a set of platform independent C++ template classes, that help to implement finite state machines as they are modeled with UML 2.1 *State Machine Diagrams*. The template classes, their attributes and operations provide a certain mapping to the UML notation elements.

The basic approach is based on the GoF **State** design pattern. The template classes are designed as base of implementation classes, that mainly concentrate on the problem domain specific functionality.

1.1 The GoF State pattern

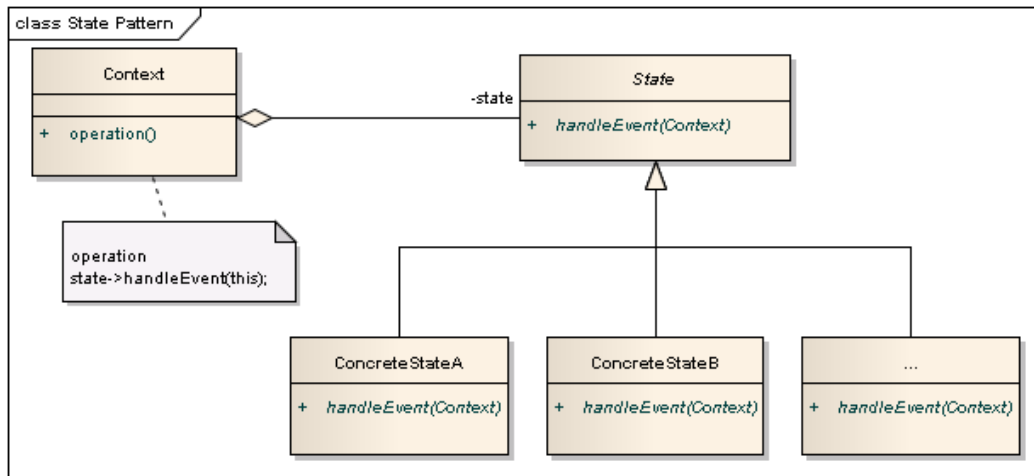
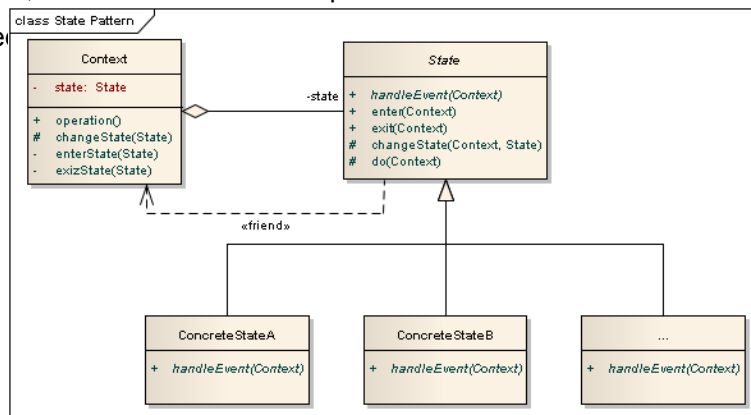


Fig 1: The structural UML representation of the GoF **State** design pattern

The class diagram in Fig 1 shows that the **Context** class exposes **operation()** to clients and internally calls one or more **handleEvent()** operations of the actual **state** member. The **State** class is abstract, and the state specific behavior is implemented in the **ConcreteStateA**, **ConcreteStateB**, ... classes. Changing the **state** reference member will change the behavior of a **Context** class object, as it would have been replaced with another class.

Fig 2 Shows a refined



a proposal for some

Fig 2: GoF State design patten implementation details

1.2 Mapping UML State Machine notation elements

Fig 3 Shows a simple *UML State Machine Diagram*, that uses basic UML State Machine Diagram notation elements.

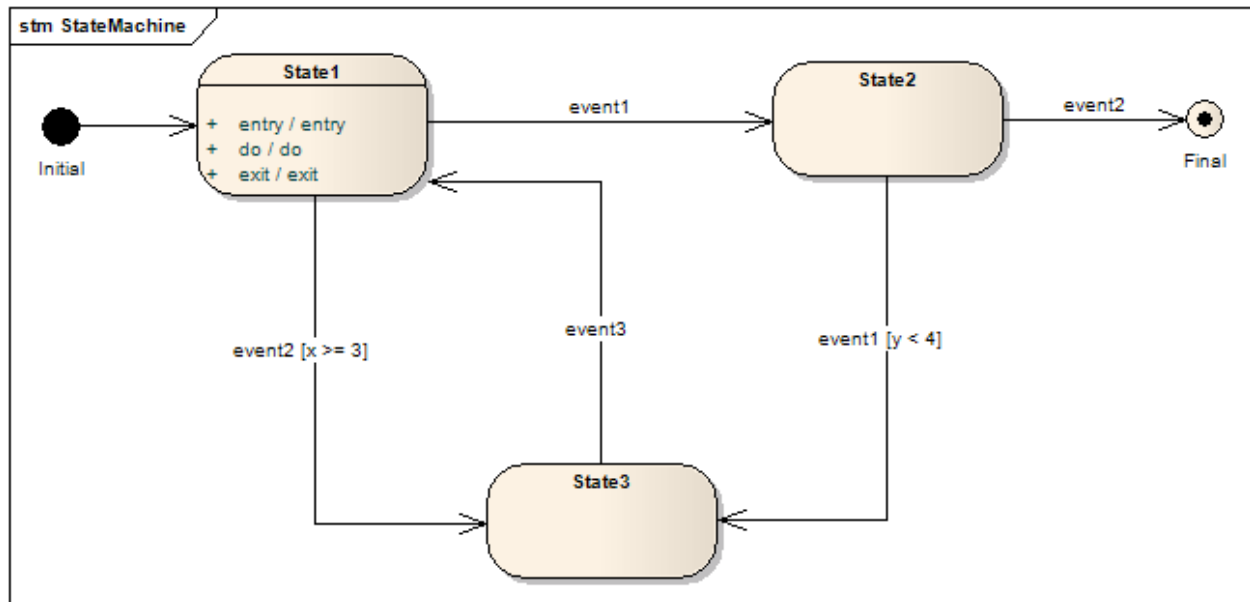
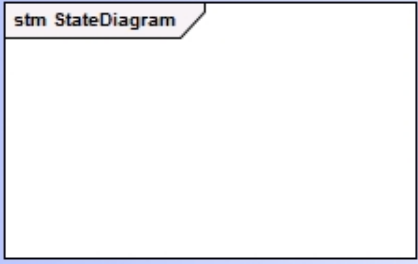
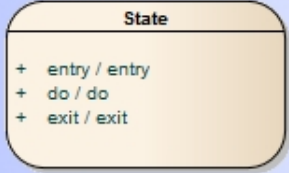
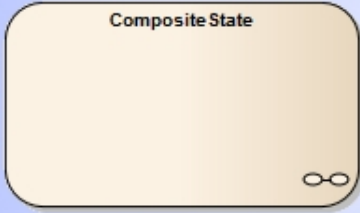
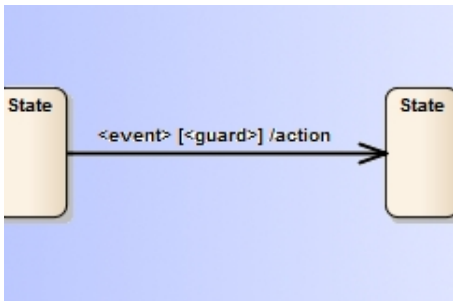


Fig 3: Simple sample UML State Machine Diagram

The following table shows how the basic *UML State Machine Diagrams* notation elements map to the elements described in the GoF *State* design pattern:

UML state machine diagram notation	GoF <i>State</i> design pattern element
 <p>The state machine diagram itself</p>	<p>A Context class instance.</p>
 <p>A state (atomic)</p>	<p>A <i>State</i> class instance (<i>ConcreteStateA</i>, <i>ConcreteStateB</i>).</p>
<p><Internal event>:= entry, do, exit, <event></p> <p>A state internal event. The do event is intrinsically triggered by the internal enter event.</p>	<p>A call to the <i>State</i> classes entry(), exit() or do() operation.</p>
 <p>A composite state. The internal states are modeled in a sub-state machine diagram</p>	<p>A <i>State</i> class instance, that also serves as another Context classes instance.</p>

UML state machine diagram notation	GoF <i>State</i> design pattern element
 <p>A transition between two states</p>	<p>A call of the Context::changeState() operation.</p>
<p><event> An event that triggers the associated transition</p>	<p>A call to a public Context::operation() operation, that delegates behavior to a State::handleEvent() operation. All events visible in the state machine diagram can be triggered via the public Context::operation() operations.</p>
<p>[guard] A conditional expression, that must return true to execute the associated transition. Guard conditions that are associated to the same source state must be mutually exclusive.</p>	<p>A conditional statement inside the Context::operation() or State::handle() operations, that decides to call Context::changeState().</p>
<p>: action, ... A list of specified event triggered operations</p>	<p>The specified action operations are called inside the implementation State::handleEvent() event handler operation¹. The calling order may be unspecified.</p>

¹⁾ Action operations that appear on a transition are not allowed to access the contexts current state. In fact these operations should be performed after the current state was exited and before the new state is entered. That's difficult to achieve with the GoF *State* design pattern, since changing state is an atomic operation in the Context class. Anyway additional behaviors can be implemented before calling the **sttcl::State<>::changeState()** operation.






UML state machine diagram notation	GoF <i>State</i> design pattern element
 <p data-bbox="646 289 946 415">Separates concurrently active regions within a composite state or state machine diagram.</p>	<p data-bbox="987 289 1430 573">This requires concurrent program execution mechanisms (e.g. threading) supported by an operating system. Each region defines an associated <i>State</i> class reference, to delegate the event handling to a <i>State::handleEvent()</i> operation concurrently.</p> <p data-bbox="987 611 1430 1079">A single region can also be considered as a concrete composite state implementation, that supports a non blocking <i>State::do()</i> operation, that executes asynchronously in a loop. Further it is necessary to have a mechanism to propagate events to the asynchronously executed operation loop. STTCL provides the <i>ConcurrentCompositeState<></i> and <i>Region<></i> template base classes to design these UML features.</p>


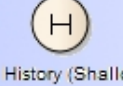
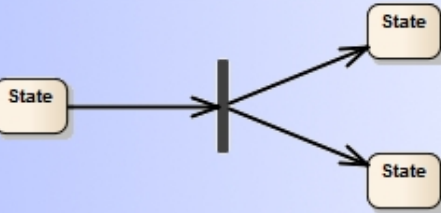
Table 1: Basic UML State Machine Elements

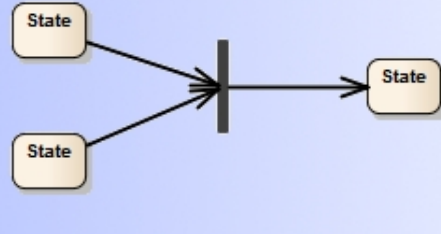
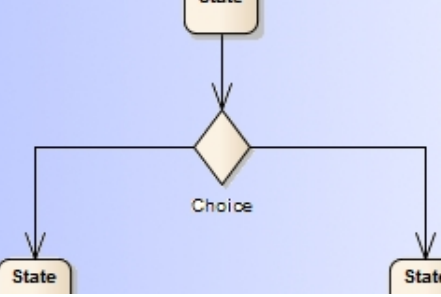
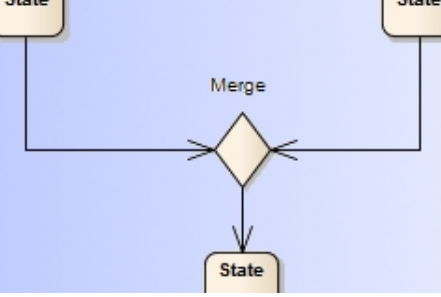
There's a number of advanced *UML State Machine Diagram* notation elements, that do not directly map to any conceptual element described in the GoF **State** design pattern. These can be mapped to certain aspects of implementation and behavior though.

This mainly concerns the so called pseudo-states, that also have incoming and/or outgoing transitions. But vs. concrete states, pseudo-states only a kind of transient states, that represent complex or intrinsic transition paths in a state machine diagram or composite state.

The following table lists possible implementation approaches for further *UML State Machine Diagram* notation elements:

UML state machine diagram notation		GoF <i>State</i> design pattern implementation
Pseudo-States		
	<p>An entry point of the state machine diagram. Initial pseudo-states never have the target role of a transition.</p>	<p>A constructor call to create a Context class instance in the simplest case. The constructor calls the Context::changeState() operation to set the initial <i>State</i> reference.</p> <p>If any events and/or guards are specified for the associated outgoing transitions, the Context class should provide a property attribute to check it's initialization status, and leave the decision, which initial <i>State</i> reference to set, to the associated Context::operation() event operations. This behavior may be encapsulated in a Context::initialize() operation.</p>
	<p>An exit point of the state machine diagram. Final pseudo-states never have the source role of a transition.</p>	<p>A <i>State</i> class implementation, that never calls the Context::changeState() operation.</p>
	<p>An exit point of a state machine or composite state triggered by the incoming transition's event.</p>	<p>A Context::finalize() operation, that calls the actual <i>State</i> references exit() operation.</p>
	<p>Exits the composite state or state machine triggered by the incoming transition's event.</p>	<p>A destructor call to a Context class instance in the simplest case.</p>

UML state machine diagram notation	GoF <i>State</i> design pattern implementation
 <p>History (Deep)</p> <p>Represents the most recent active configuration of a composite state. In opposite to the shallow history pseudo-state, this includes all sub states of all regions and their recently active sub states recursively.</p>	<p>The composite Context class instance must keep track of the most recent sub <i>State</i> reference, when the composite <i>State</i> classes exit() operation is called. When the composite <i>State</i> classes enter() operation is called later on, the composite Context class directly transits to the remembered most recent sub <i>State</i> reference.</p> <p>STTCL composite state classes provide the <i>HistoryType</i> template parameter to determine the history behavior.</p>
 <p>History (Shallow)</p> <p>Represents the most recent sub state of a composite state. A composite state can have at most one history pseudo-state. At most one transition to the default sub state may originate the history pseudo-state. This transition is executed in case the composite state was never active before.</p>	<p>The same behavior as described for the deep History. But In case, that a reentered sub <i>State</i> reference also represents a composite state, it's composite Context class must be (re-)initialized.</p> <p>STTCL composite state classes provide the <i>HistoryType</i> template parameter to determine the history behavior.</p>
 <p>A Fork. Serves to split a single incoming transition into concurrently executed outgoing transitions. No guards are allowed on any associated transitions.</p>	<p>A fork represents the initiation of concurrently executed operations (i.e. tasks, threads) of a composite Context class. This can be implemented as a non blocking operation that starts all of the associated concurrently executed operations.</p>

UML state machine diagram notation	GoF <i>State</i> design pattern implementation
 <p>A Join. Serves to synchronize multiple concurrently executed incoming transitions into a single outgoing transitions. No guards are allowed on any associated transitions.</p>	<p>A join represents a synchronization point (i.e. semaphore, mutex) for formerly initiated concurrently executed operations of a composite <i>Context</i> class. This can be implemented in a blocking operation, that waits on completion of all the associated concurrently executed operations.</p>
 <p>A Choice. Serves to select the outgoing transitions according runtime conditions represented by the guards, associated to them. The guard conditions must be mutually exclusive, to choose a certain transition path. The model requires at least one of the guard conditions to evaluate to true, therefore one of the outgoing transitions should cover the else/default case. Unlike the Fork pseudo-state, a Choice node doesn't initiate any concurrently executed transitions.</p>	<p>A Choice can be implemented as a <code>if ...else if ..else</code> or <code>switch</code> conditional block in a <i>State</i> classes implementation (<i>ConcreteStateA</i>, <i>ConcreteStateB</i>), that choose the appropriate target <i>State</i> reference parameter for a call to the <code>Context::changeState()</code> operation.</p> <p>Note: Choices don't serve to split transition paths into concurrently executed operations!</p>
 <p>A Merge. Serves to combine alternate execution flows into a single outgoing transition. Unlike the Join pseudo-state, a Merge node doesn't provide synchronization of concurrently executed transitions, that originate from different regions. Also a Merges incoming transitions may have guard conditions associated.</p>	<p>A Merge can be implemented as an operation, that is shared by a number of <i>State</i> class implementations (<i>ConcreteStateA</i>, <i>ConcreteStateB</i>), and ends up in a single call of <code>Context::changeState()</code> operation. The decision to call this operation is done in the implementation of the <code>State::handleEvent()</code> operation, according the associated guard condition.</p>

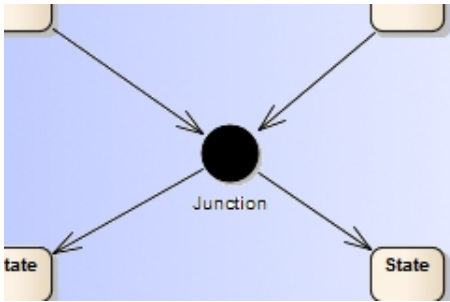
UML state machine diagram notation	GoF <i>State</i> design pattern implementation
 <p>A Junction. Serves to share transition paths for the incoming transitions. The incoming and or outgoing transitions have guard conditions associated. Incoming transitions are shared between the source states. Outgoing transitions must have mutually exclusive guard conditions. The model requires at least one of the guard conditions to evaluate to true, therefore one of the outgoing transitions should cover the else/default case.</p>	<p>The Junction serves complex conditional path transitions that can be implemented in a similar way as the Choice and Merge pseudo-states.</p> <p>UML 2.1 specification restricts the guard conditions to be static (actively waiting for all incoming events). IMHO this can be interpreted, that the Junction node should be another implementation of the <i>State</i> class. Such implementation should not affect the <i>Context</i> classes attributes, but just serve to forward incoming events to outgoing transitions (i.e. <i>Context::changeState()</i> calls).</p>

Table 2: Advanced UML State Machine Elements

2 Implementation Design

The basic implementation approach of the C++ **STTCL** is, to provide abstractions of the GoF **State** design patterns static structures. This is accomplished using parameterized base classes that serve certain functionality, associated to the formerly listed *UML State Machine Diagram* notation elements.

The **Context::operation()** and **State::handleEvent()** operations, described in the design pattern, can be considered as a pair of compliant, application specific interfaces (source/sink).

The interface realized by the **Context** class is visible to any clients of the state machine implementation. The interface realized by the **State** class implementations, and those implementations themselves, shouldn't be visible to any clients of the state machine implementation.

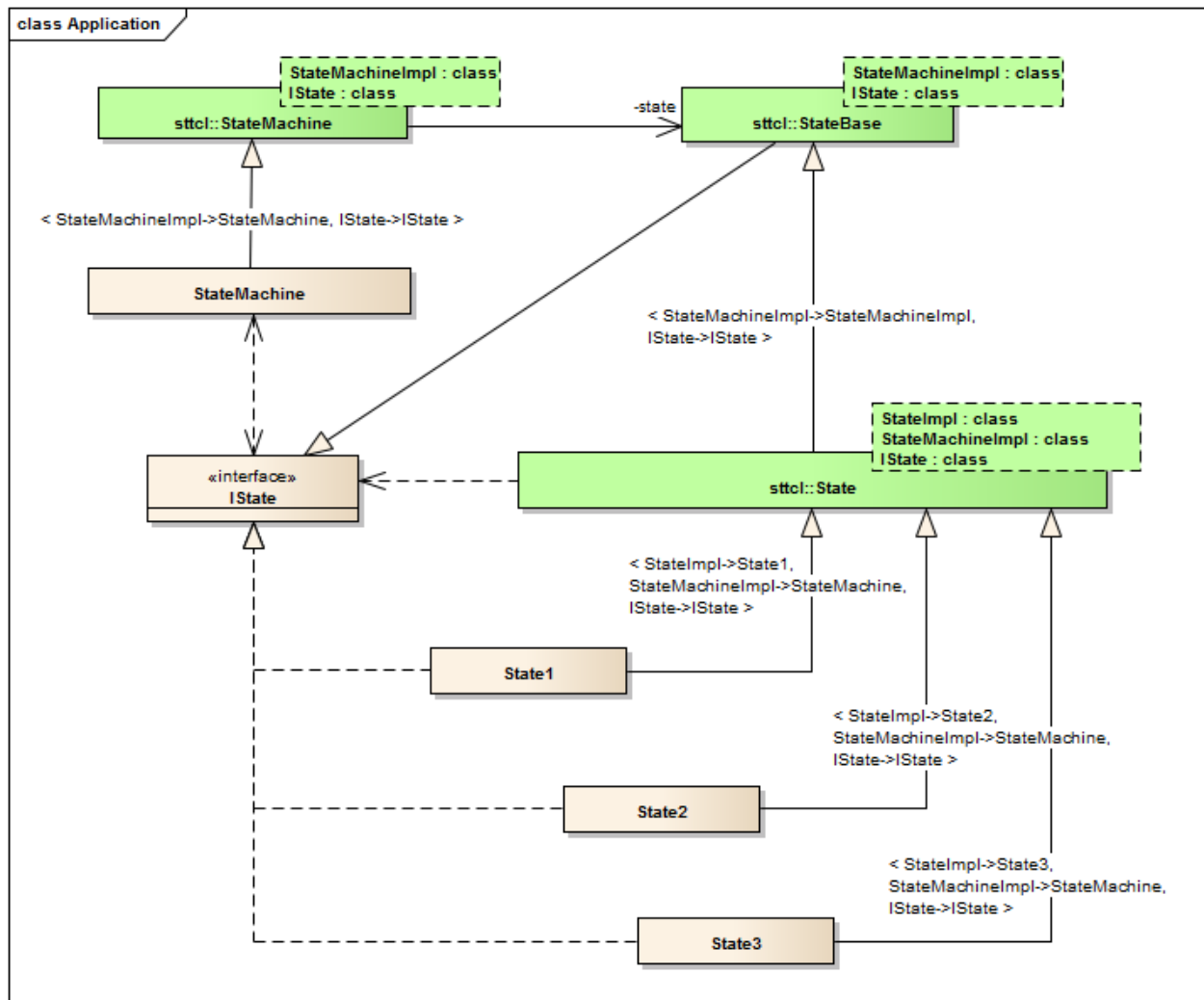


Fig 4: The basic STTCL **State** design pattern abstraction

Fig 4 Shows a class diagram, that illustrates a state machine implementation that uses the STTCL basic classes. The highlighted elements represent the application specific stuff.

The `State<>` template base class will provide certain common state specific operations, like `entry()`, `exit()` and `do()`. These operations are not intended to be called by the implementation classes, but rather by the corresponding `StateMachine` class. The `State` class also provides a protected operation `changeState()`, that will delegate to a call to the `StateMachineImpl` context parameters `StateMachine<>::changeState()` operation. This operation enables the concrete state implementations, to implement transitions to another concrete state. As formerly stated, the `StateMachine<>::changeState()` operation shouldn't be directly accessible for any client (or actor) classes of the state machine. This requires, that the `State<>::changeState()` operation is allowed to friendly access the `StateMachine<>::changeState()` operation.

The `StateMachine<>` class mainly serves to implement the transitions' behavior, when they are enabled and passed the guard conditions implemented in a concrete state. This concerns control of the exited and entered state's synchronous and asynchronous execution behavior.

As discussed in the GoF *State* design pattern, the concrete state implementations may provide singleton instances (accessible though a static operation of the class), as far no state runtime attributes need to be maintained. This approach will guarantee, that the `StateMachine<>` implementation doesn't need to reference concrete `State<>` implementations, other than it's initial state.

2.1 Aspect Oriented Modelling of State Machines

STTCL uses aspect oriented design for certain aspects of UML 2.2 state diagram notation elements. The different aspect *variations* are selected through template parameters and concern the following:

- Composite states (aka HSM, hierarchical state machines)
 - State history behavior
- Concurrency
 - Active states
 - State Machine regions
- Direct transitions

2.1.1 Composite state aspect

A composite state can be implemented as a merge of `sttcl::StateMachine<>` and `sttcl::State<>` class. But composite states also introduce the aspect of state history behaviour. There are three behaviors defined by UML 2.2:

- No state history, the sub state machine is initialized every time, when the composite state is entered.
- Deep state history, the sub state machine directly transits to the last remembered state, this is applied recursively for further contained composite states.
- Shallow state history, the sub state machine directly transits to the last remembered state, further contained composite states are initialized.

STTCL provides the `sttcl::CompositeState<>` and `sttcl::Region<>` template base classes that support selecting a state history behavior implementation. As value for the *HistoryType* template parameter choose one of the `sttcl::CompositeStateHistoryType` enum values *Deep* or *Shallow* to implement a pseudo-state for the history, or *None* to have no pseudo-state.

2.1.2 *Concurrency aspect*

UML 2.2 state diagrams have several notation elements that model orthogonal state transition paths. That means that operations that appear on forked transition paths should be executed concurrently in separate threads. Also state internal *do* actions may execute “in background” as long the state is active (though UML 2.2 does not specify a special notation for that case).

The latter aspect is supported by the `sttc1::ActiveState<>` template base class. The `ActiveState` class implements a background thread loop, that calls a specified *do* operation. The *do* operation is called either once, cyclically with a specified frequency or cyclically non blocking.

Orthogonal states and transition paths are supported by the `sttc1::ConcurrentCompositeState<>` and `sttc1::Region<>` template base classes.

The `ConcurrentCompositeState` class effectively implements a fork to all contained `Region` composite states, and dispatches incoming event triggers to the region threads.

To dispatch events in the `Region` template base class, the `sttc1::Sttc1EventQueue<>` class is used by default, you may specify your own event queue implementation using the *EventQueueType* template parameter.

2.1.3 *Direct transition aspect*

With UML 2.2 transition notation event triggers are optional. If a transition has no triggers we'll call it a direct transition here (the UML 2.2 spec calls these *Completion transitions*). Following the GoF state pattern, state transitions are usually triggered using the event handler methods of the state interfaces, but with direct transitions there's no such event handler method available.

The UML 2.2 superstructure specification says in section '15.3.14 Transition (from BehaviorStateMachines)':

Completion transitions and completion events

A completion transition is a transition originating from a state or an exit point but which does not have an explicit trigger, although it may have a guard defined. A completion transition is implicitly triggered by a completion event. In case of a leaf state, a completion event is generated once the entry actions and the internal activities (“do” activities) have been completed. If no actions or activities exist, the completion event is generated upon entering the state. If the state is a composite state or a submachine state, a completion event is generated if either the submachine or the contained region has reached a final state and the state’s internal activities have been completed. This event is the implicit trigger for a completion transition. The completion event is dispatched before any other events in the pool and has no associated parameters. For instance, a completion transition emanating from an orthogonal composite state will be taken automatically as soon as all the orthogonal regions have reached their final state.

If multiple completion transitions are defined for a state, then they should have mutually exclusive guard conditions.

This means effectively that direct transitions are implicitly triggered whenever the state *do* actions are completed. The `sttc1::State<>` and `sttc1::ActiveState<>` template base classes provide the

overridable method `checkDirectTransitionImpl()` that is called directly after the `doAction` (if there is any) returns. This callback method should return the next state that is connected with any direct transitions, if null is returned there's no direct transition executed. Additionally you can specify if there's a direct transition going to the containing state machine's final pseudo-state, a null must be returned for the next state in this case.

Checking guard conditions should naturally appear in the overridden implementation of this method.

2.2 Configuring the STTCL library

The concurrency features need some OS specific implementation for threads, mutexes, semaphores and “real” timing capabilities.

STTCL provides builtin concurrency support for certain build environments, currently no OS specific implementations are available. The builtin environments are:

- boost, using the boost/thread, boost/interprocess and boost/date_time libraries
- POSIX, using the pthread library and POSIX time API
- c++11, using the C++ 11 standard library functions

Additionally some implementations rely on the C++ standard template libraries by default. Currently the following STL classes are used:

- `std::deque<T>`

2.2.1 Configuring STTCL STL dependency

Some classes in the STTCL library use the C++ STL for implementation of container classes and other standard constructs. To enable the use of the STL classes you must set the `STTCL_USE_STL` define (add `-DSTTCL_USE_STL` to your compiler flags).

If you want to introduce your own implementations omit the `STTCL_USE_STL` define and define the following macros to replace particular STL default classes:

```
#define STTCL_DEFAULT_DEQUEIMPL(__T__) MyDequeImpl<__T__>
```

2.2.2 Configuring STTCL builtin concurrency implementations

STTCL uses wrapper classes (adapters) for the environment specific implementations of the above mentioned capabilities:

- [`sttcl::SttclThread<>`](#) as thread adapter
- [`sttcl::SttclMutex<>`](#) as mutex abstraction (needs timed/unblocking try_lock() implementation)
- [`sttcl::SttclSemaphore<>`](#) as semaphore abstraction (needs timed/unblocking try_wait() implementation)
- [`sttcl::TimeDuration<>`](#) as abstraction for a “real”-time duration
- [`sttcl::SttclEventQueue<>`](#) as abstraction for a thread safe event queue

To use the builtin implementations you need to build the STTCL source files using one of the following defines (add `-D<config>` to your compiler flags):

- `STTCL_BOOST_IMPL` to select the boost implementation as default
- `STTCL_POSIX_IMPL` to select the POSIX implementation as default
- `STTCL_CX11_IMPL` to select the C++ 11 standard implementation as default

2.2.3 *Providing custom implementations for concurrency*

You may implement your own abstractions for threads, mutexes, semaphores and time duration representation. Provide the following defines to set your custom implementation as defaults (these must be seen by the STTCL header files):

```
#define STTCL_DEFAULT_THREADIMPL MyThreadImpl  
#define STTCL_DEFAULT_MUTEXIMPL MyMutexImpl  
#define STTCL_DEFAULT_SEMAPHOREIMPL MySemaphoreImpl  
#define STTCL_DEFAULT_TIMEDURATIONIMPL MyTimeDurationImpl
```

.Alternatively you can provide your implementations directly as template parameters of the STTCL template base classes.

2.3 STTCL Base Classes

The STTCL template base classes provide default implementations for standard state machine context and state behavior. The behavioral aspects may be overridden by the implementation class that is passed as template parameter. The methods called to implement behavioral aspects are designed as implementation hooks, such that the base class implements a default behavior and the specific method is called using a static cast to the implementation class.

2.3.1 *sttcl::StateMachine<>*

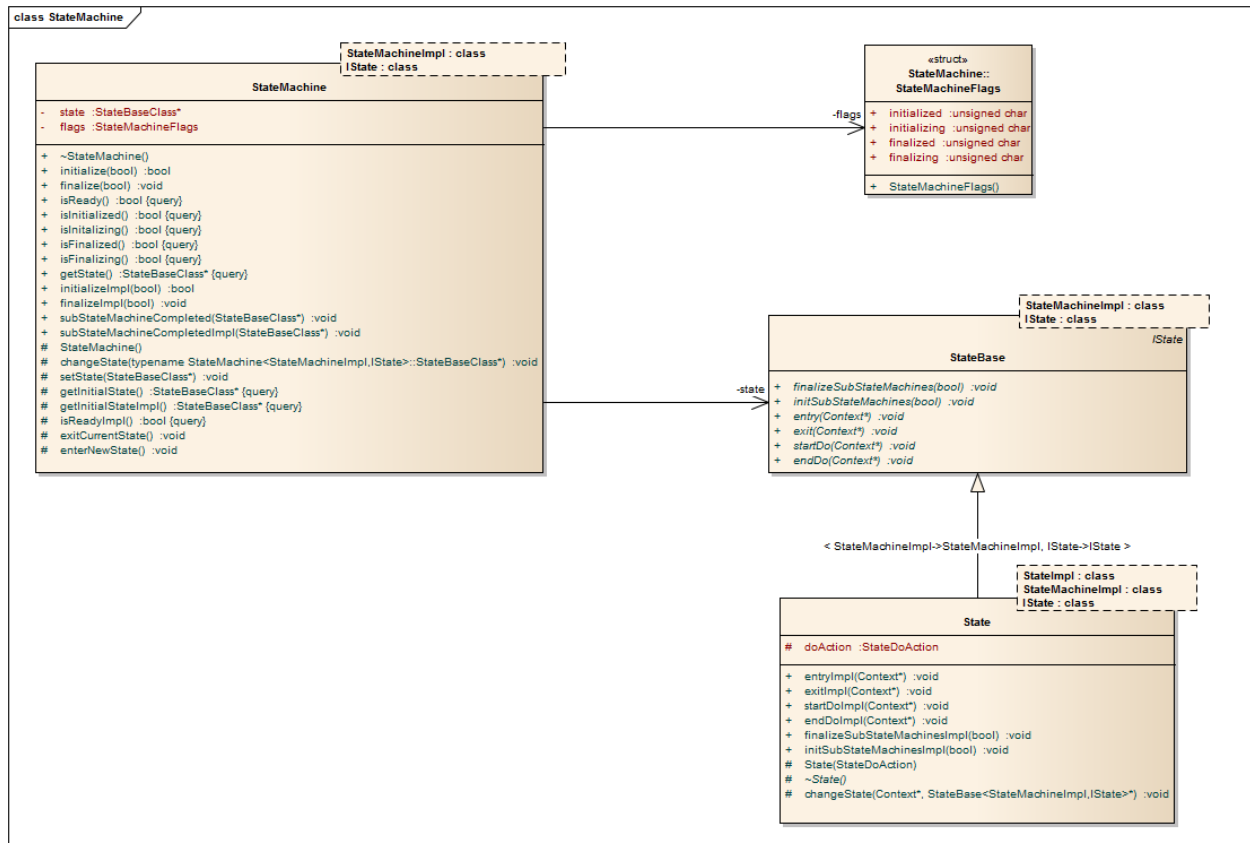


Fig 5: `sttcl::StateMachine<>` class diagram

The **StateMachine** template class serves as base class for the top level state machine implementation. The **IState** template parameter specifies the event handler interface of the **State** implementation classes to appear in the state diagram. This interface can be called in the **StateMachine** implementation, to realize triggering events.

Template signature:

```
template<class StateMachineImpl, class IState>
class StateMachine;
```

StateMachineImpl specifies the inheriting class.
IState specifies the internal state interface class.

The **StateMachine<>** template base class implements the following main operations:

+initialize()

Sets the state machine to its initial state.

+finalize()

Exits the state machines current state and resets the state machine.

+getState()

Gets the state machines current state.

#changeState()

Changes the state machines current state.

Implementation hooks:

+initializeImpl()

Overrides the default initialize() behavior. An override should call the default implementation.

+finalizeImpl()

Overrides the default finalize() behavior. An override should call the default implementation.

+getInitialStateImpl()

Must be implemented. Returns the initial state of the state machine implementation.

2.3.2 sttcl::State<>

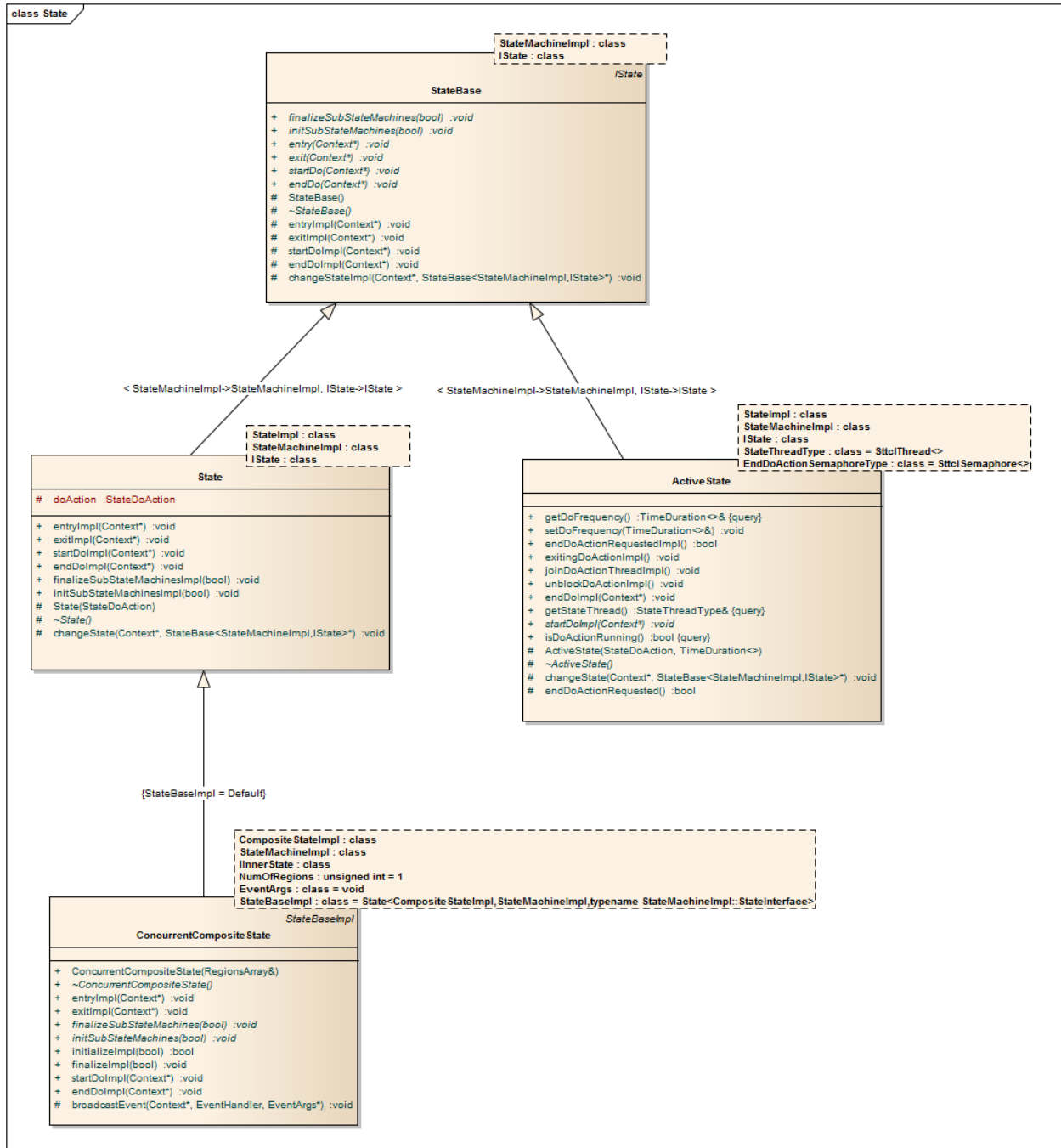


Fig 6: sttcl::State<>, sttcl::ActiveState<>, sttcl::ConcurrentCompositeState<> class diagram

The **State** template class serves as base class for any state that appears in the state machine implementation specified with the **StateMachineImpl** parameter. Implementation classes must

implement the event handler interface specified with the *IState* template parameter.

Template signature:

```
template<class StateImpl, class StateMachineImpl, class IState>
class State;
```

StateImpl specifies the inheriting class.

StateMachineImpl specifies the containing state machine implementation.

IState specifies the internal state interface class.

The `sttcl::State<>` template base class implements the following main operations:

-**entry()**

Called when the state is entered.

-**startDo()**

Called when the state's do action is called.

-**endDo()**

Called when the state's do action should be terminated.

-**exit()**

Called when the state is left.

-**initSubStateMachines()**

Called to initialize a state's sub state machines.

-**finalizeSubStateMachines()**

Called to finalize a state's sub state machines.

Implementation hooks:

+**entryImpl()**

Overrides the default entry() behavior. An override should call the default implementation.

+**startDoImpl()**

Overrides the default startDo() behavior. An override should call the default implementation.

+**endDoImpl()**

Overrides the default endDo() behavior. An override should call the default implementation.

+**exitImpl()**

Overrides the default exit() behavior. An override should call the default implementation.

+**initSubStateMachinesImpl()**

Overrides the default initSubStateMachines() behavior. An override should call the default implementation.

+**finalizeSubStateMachinesImpl()**

Overrides the default finalizeSubStateMachines() behavior. An override should call the default implementation.

+**checkDirectTransitionImpl()**

Overrides the default checkDirectTransitionImpl() behavior. An override should call the default implementation.

2.3.3 *sttcl::ActiveState<>*

The *ActiveState* template class serves as base class for any state with a asynchronously executing do action that appears in the state machine implementation specified with the *StateMachineImpl* parameter. Implementation classes must implement the event handler interface specified with the *IState* template parameter.

The do action that is specified in the constructor can either be executed once, with a particular

frequency or may use its own synchronization mechanisms for asynchronous execution.

Template signature:

```
template
< class StateImpl
, class StateMachineImpl
, class IState
, class StateThreadType
, class TimeDurationType
, class EndDoActionSemaphoreType
, class ActiveStateMutexType
>
```

```
class ActiveState;
```

StateImpl specifies the inheriting class.

StateMachineImpl specifies the containing state machine implementation.

IState specifies the internal state interface class.

StateThreadType specifies the thread class to implement the internal state thread.

TimeDurationType specifies the time duration representation implementation class.

EndDoActionSemaphoreType specifies the semaphore class to implement the semaphore that is used to signal termination of the internal state thread.

ActiveStateMutexType specifies the mutex class used to provide thread safe access to the **sttcl::ActiveState<>** class member variables.

The **sttcl::ActiveState<>** template base class implements the following main operations:

-entry()

Called when the state is entered.

-startDo()

Called when the state's do action is called.

-endDo()

Called when the state's do action should be terminated.

-exit()

Called when the state is left.

-initSubStateMachines()

Called to initialize a state's sub state machines.

-finalizeSubStateMachines()

Called to finalize a state's sub state machines.

Implementation hooks:

+entryImpl()

Overrides the default entry() behavior. An override should call the default implementation.

+startDoImpl()

Overrides the default startDo() behavior. An override should call the default implementation.

+endDoImpl()

Overrides the default endDo() behavior. An override should call the default implementation.

+exitImpl()

Overrides the default exit() behavior. An override should call the default implementation.

+initSubStateMachinesImpl()

Overrides the default initSubStateMachines() behavior. An override should call the default

implementation.

+finalizeSubStateMachinesImpl()

Overrides the default finalizeSubStateMachines() behavior. An override should call the default implementation.

+checkDirectTransitionImpl()

Overrides the default checkDirectTransitionImpl() behavior. An override should call the default implementation.

+exitingDoActionImpl()

Overrides the default exitingDoActionImpl() behavior. This callback signals that the do action execution has finished (*Completion event*).

+joinDoActionThreadImpl()

Overrides the default joinDoActionImpl() behavior. This callback implements the synchronization point for the internal do action thread and the states outgoing transition. An override should call the default implementation.

+unblockDoActionImpl()

Overrides the default unblockDoActionImpl() behavior. This callback allows an implementation to unblock the internal do action thread and proceed with the states outgoing transition.

StateMachineImpl specifies the containing state machine implementation.

InnerState specifies the internal state interface class.

HistoryType optionally specifies a history pseudo-state.

StateBaseImpl optionally specifies a `sttcl::State<>` implementation base class.

StateMachineBaseImpl optionally specifies a `sttcl::StateMachine<>` implementation base class.

The `sttcl::CompositeState<>` template base class implements the following main operations:

#initSubStateMachines()

Called to initialize a state's sub state machines.

#finalizeSubStateMachines()

Called to finalize a state's sub state machines.

#changeState()

Called to change the containing state machines next state.

+changeState()

Called to change the internal state machines next state.

+subStateMachineCompleted()

Called to notify the implementation class that the CompositeState state machine is finalized.

Additionally the main operations of the *StateBaseImpl* and *StateMachineBaseImpl* template parameter classes are inherited.

Implementation hooks:

+entryImpl()

Overrides the default entry() behavior. An override should call the default implementation.

+startDoImpl()

Overrides the default startDo() behavior. An override should call the default implementation.

+endDoImpl()

Overrides the default endDo() behavior. An override should call the default implementation.

+exitImpl()

Overrides the default exit() behavior. An override should call the default implementation.

+initSubStateMachinesImpl()

Overrides the default initSubStateMachines() behavior. An override should call the default implementation.

+finalizeSubStateMachinesImpl()

Overrides the default finalizeSubStateMachines() behavior. An override should call the default implementation.

+checkDirectTransitionImpl()

Overrides the default checkDirectTransitionImpl() behavior. An override should call the default implementation.

+subStateMachineCompletedImpl()

Overrides the default subStateMachineCompleted() behavior. An override should call the default implementation.

2.3.5 sttcl::ConcurrentCompositeState<>

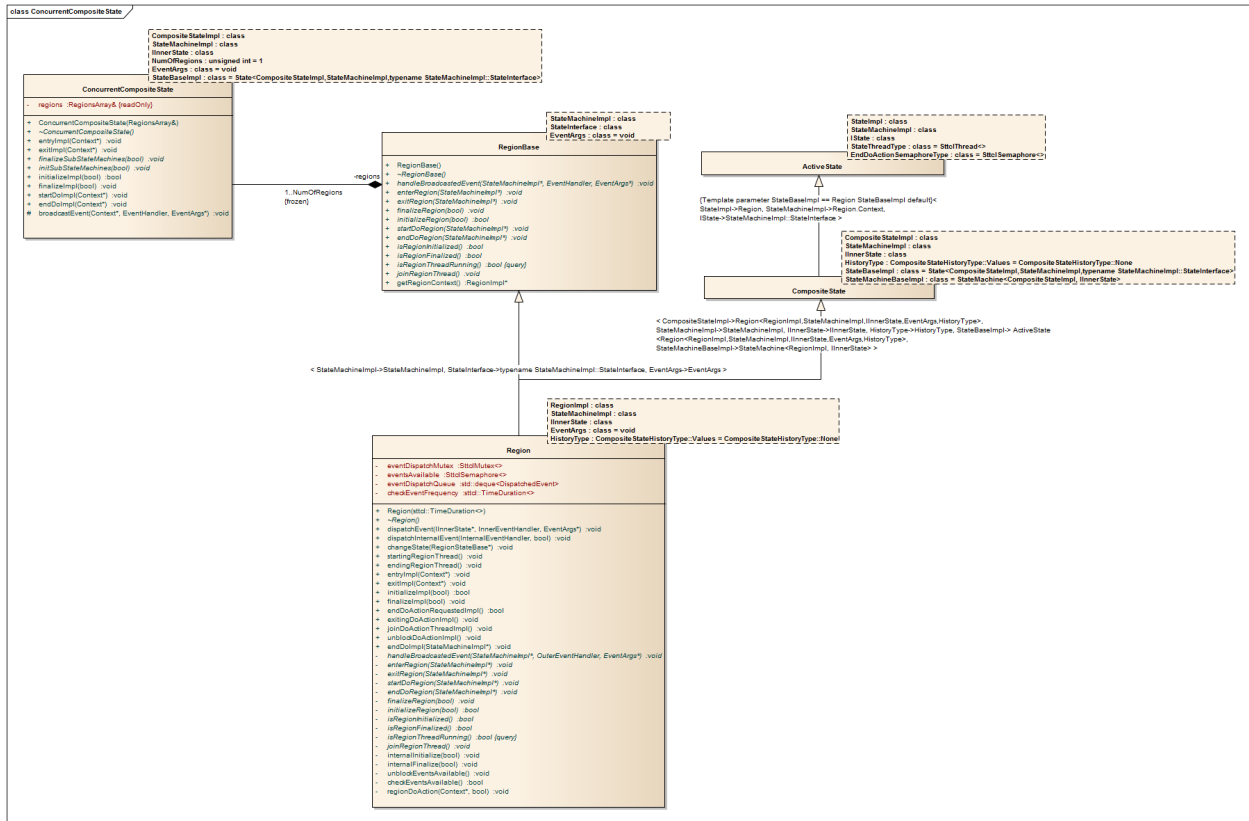


Fig 8: Class diagram for `sttcl::ConcurrentCompositeState<>` and `Region<>`

The `ConcurrentCompositeState` template class serves as base class for a composite state that contains `sttcl::Region<>` implementations with orthogonal states. The implementation class constructor must pass an array of `sttcl::RegionBase<>*` pointers to the `ConcurrentCompositeState` constructor. The array must have the exact size as specified with the `NumOfRegions` template parameter.

Template signature:

```

template
< class CompositeStateImpl
, class StateMachineImpl
, class IInnerState
, unsigned int NumOfRegions
, class EventArgs
, class StateBaseImpl
>
class ConcurrentCompositeState;

```

`StateImpl` specifies the inheriting class.

StateMachineImpl specifies the containing state machine implementation.

IInnerState specifies the internal state interface class.

NumOfRegions specifies the number of contained `sttcl::Region<>` implementation class instances.

EventArgs optionally specifies a class that is passed through the **IInnerState** interface methods.

StateBaseImpl optionally specifies a `sttcl::State<>` implementation base class.

The `sttcl::ConcurrentCompositeState<>` template base class implements the following main operations:

initSubStateMachines()

Called to initialize a state's sub state machines.

finalizeSubStateMachines()

Called to finalize a state's sub state machines.

#broadcastEvent()

Called to queue events to the internal threads of the contained `sttcl::Region<>` implementation class instances.

Additionally the main operations of the **StateBaseImpl** template parameter class are inherited.

Implementation hooks:

+entryImpl()

Overrides the default entry() behavior. An override should call the default implementation.

+startDoImpl()

Overrides the default startDo() behavior. An override should call the default implementation.

+endDoImpl()

Overrides the default endDo() behavior. An override should call the default implementation.

+exitImpl()

Overrides the default exit() behavior. An override should call the default implementation.

+initializeImpl()

Overrides the default initialize() behavior. An override should call the default implementation.

+finalizeImpl()

Overrides the default finalize() behavior. An override should call the default implementation.

+checkDirectTransitionImpl()

Overrides the default checkDirectTransitionImpl() behavior. An override should call the default implementation.

2.3.6 `sttcl::Region<>`

The **Region** template class serves as a base class for a region sub statemachine implementation. As the diagram in Fig 8 shows the **Region** class inherits from both `sttcl::CompositeState<>` and the `sttcl::ActiveState<>` class (as CompositeState **StateBaseImpl**) and thus their behaviors.

Template signature:

template

< class RegionImpl

, class StateMachineImpl

, class IInnerState

, class EventArgs

, sttcl::CompositeStateHistoryType::Values HistoryType

, class StateThreadType

```
, class TimeDurationType
, class SemaphoreType
, class MutexType
, class EventQueueType
> class Region;
```

StateImpl specifies the inheriting class.

StateMachineImpl specifies the containing state machine implementation.

InnerState specifies the internal state interface class.

EventArgs optionally specifies a class that is passed through the **InnerState** interface methods.

HistoryType optionally specifies a history pseudo-state.

StateThreadType specifies the thread class to implement the internal state thread.

TimeDurationType specifies the time duration representation implementation class.

EndDoActionSemaphoreType specifies the semaphore class to implement the semaphore that is used to signal termination of the internal state thread.

ActiveStateMutexType specifies the mutex class used to provide thread safe access to the **sttcl::ActiveState<>** class member variables.

The **sttcl::Region<>** template base class implements the following main operations:

#initSubStateMachines()

Called to initialize a state's sub state machines.

#finalizeSubStateMachines()

Called to finalize a state's sub state machines.

#changeState()

Called to change the containing state machines next state.

+changeState()

Called to change the internal state machines next state.

+subStateMachineCompleted()

Called to notify the implementation class that the CompositeState state machine is finalized.

Additionally the main operations of the **StateBaseImpl** and **StateMachineBaseImpl** template parameter classes are inherited.

Implementation hooks:

+entryImpl()

Overrides the default entry() behavior. An override should call the default implementation.

+startDoImpl()

Overrides the default startDo() behavior. An override should call the default implementation.

+endDoImpl()

Overrides the default endDo() behavior. An override should call the default implementation.

+exitImpl()

Overrides the default exit() behavior. An override should call the default implementation.

+initSubStateMachinesImpl()

Overrides the default initSubStateMachines() behavior. An override should call the default implementation.

+finalizeSubStateMachinesImpl()

Overrides the default finalizeSubStateMachines() behavior. An override should call the default implementation.

+checkDirectTransitionImpl()

Overrides the default `checkDirectTransitionImpl()` behavior. An override should call the default implementation.

+subStateMachineCompletedImpl()

Overrides the default `subStateMachineCompleted()` behavior. An override should call the default implementation.

+exitingDoActionImpl()

Overrides the default `exitingDoActionImpl()` behavior. This callback signals that the do action execution has finished (*Completion event*).

+joinDoActionThreadImpl()

Overrides the default `joinDoActionImpl()` behavior. This callback implements the synchronization point for the internal do action thread and the states outgoing transition. An override should call the default implementation.

+unblockDoActionImpl()

Overrides the default `unblockDoActionImpl()` behavior. This callback allows an implementation to unblock the internal do action thread and proceed with the states outgoing transition.

2.4 STTCL configuration adapters

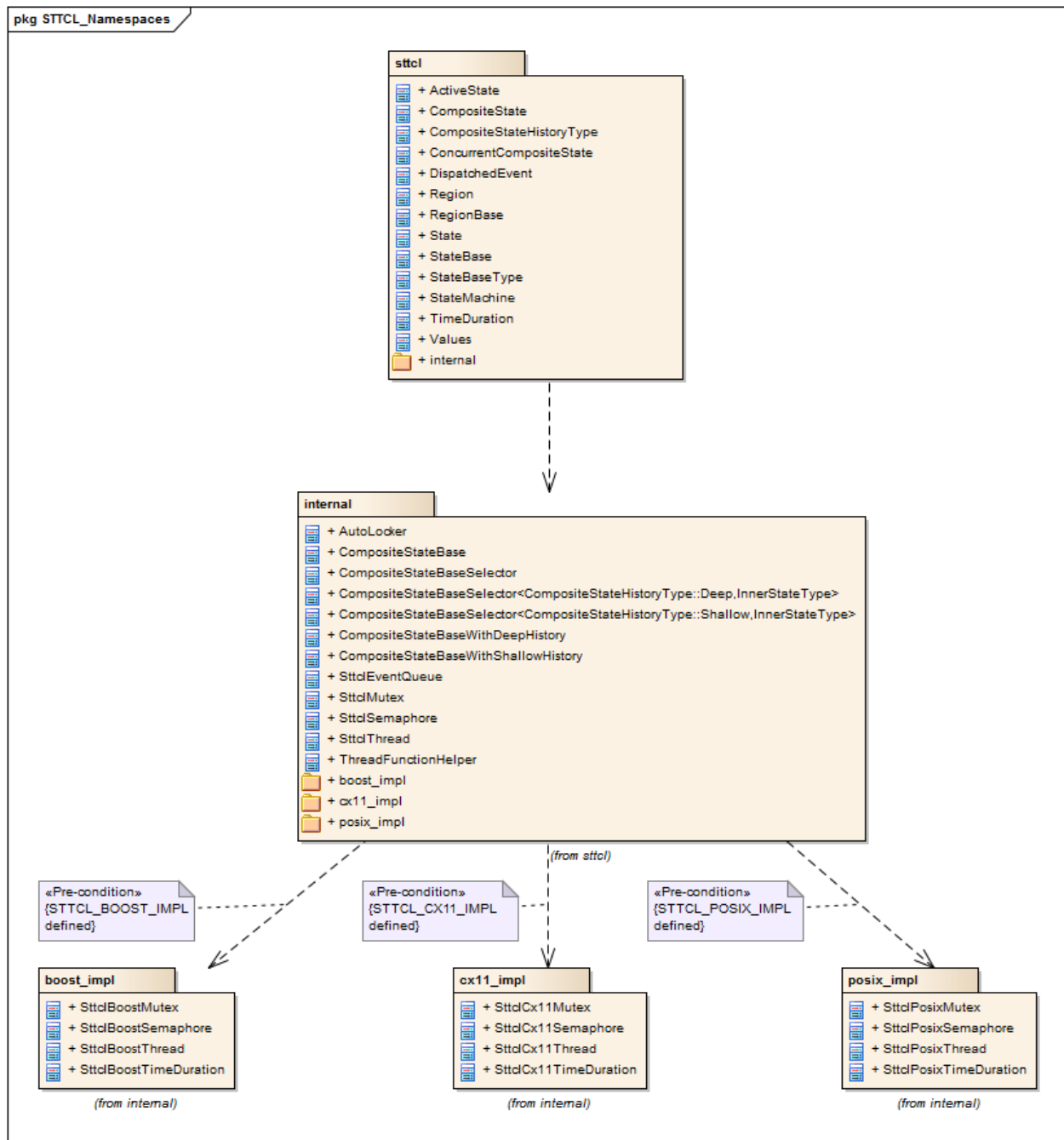


Fig 9: Package diagram for STTCL namespaces

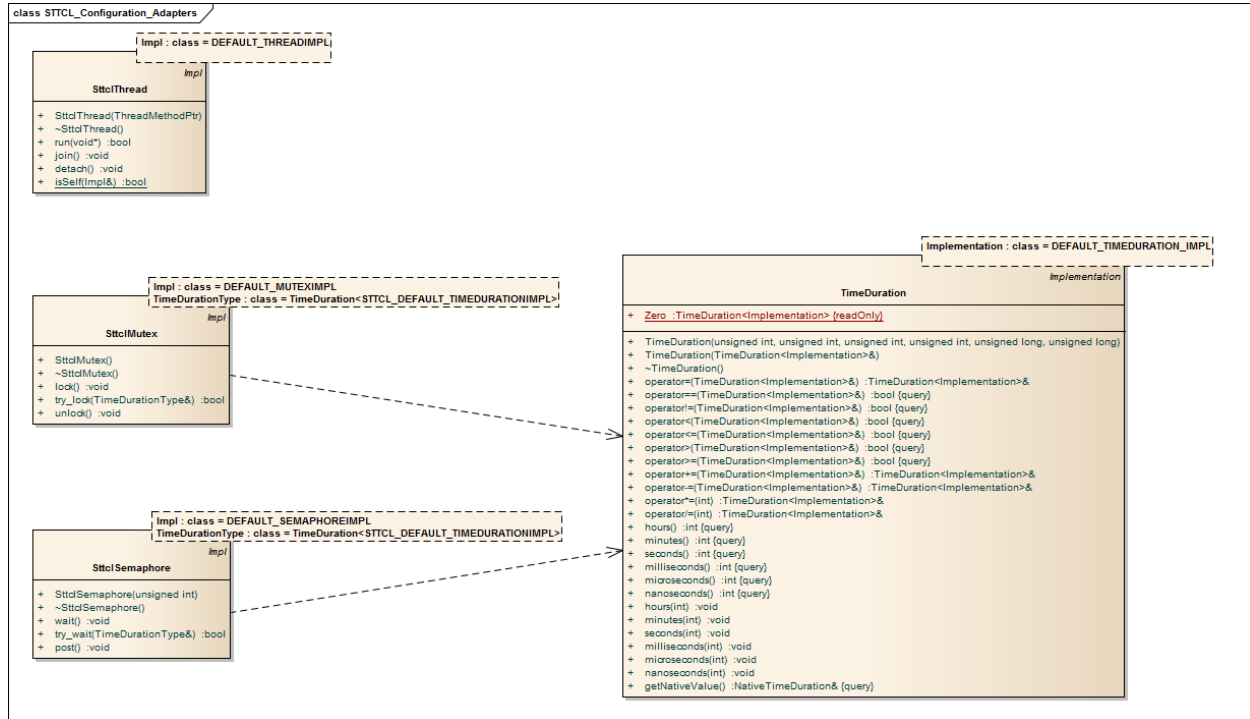


Fig 10: Class diagram `sttcl::SttclThread<>`, `sttcl::SttclMutex<>`, `sttcl::SttclSemaphore<>` and `sttcl::TimeDuration<>`

The configuration adapter classes are wrappers for build environment/OS specific abstractions used by the STTCL template base classes. The optional **Impl** template parameter specifies the concrete implementation base class of the wrapper class and can be configured generally using one of the corresponding macro definitions described in 2.2.3 Providing custom implementations for concurrency.

2.4.1 `sttcl::internal::SttclThread<>`

The **SttclThread** class declares an abstraction for a thread implementation class.

Template signature:

```
template<class Impl = STTCL_DEFAULT_THREADIMPL>
class SttclThread;
```

Impl specifies the implementation class.

Type definitions:

```
typedef void* (*ThreadMethodPtr)(void*);
```

Required implementation interface:

```
Impl(ThreadMethodPtr argThreadMethod)
```

Constructor for implementation class.

```
bool run(void* args);
```

Runs the method passed in the constructor in a separate thread.

```
void join();
```

Waits blocking forever until the thread exits.


```
void detach();
```

Kills the thread.

```
static bool isSelf(const Impl& otherThread);
```

Checks if the calling method runs within the thread specified with the *otherThread* parameter.

2.4.2 *sttcl::internal::SttclMutex<>*

The *SttclMutex* class declares an abstraction for a mutex implementation class.

Template signature:

```
template
< class Impl = STTCL_DEFAULT_MUTEXIMPL
, class TimeDurationType = TimeDuration<STTCL_DEFAULT_TIMEDURATIONIMPL>
>
class SttclMutex;
```

Impl specifies the implementation class.

TimeDurationType specifies the time duration representation class to use.

Required implementation interface:

```
Impl()
```

Default constructor for implementation class.

```
void Lock();
```

Locks the mutex. Waits blocking forever until the mutex becomes lockable.

```
bool try_lock(const TimeDurationType& timeout);
```

Tries to lock the mutex within the specified timeout parameter.

```
void unlock();
```

Unlocks the mutex.

2.4.3 *sttcl::internal::SttclSemaphore<>*

The *SttclSemaphore* class declares an abstraction for a semaphore implementation class.

Template signature:

```
template
< class Impl = STTCL_DEFAULT_MUTEXIMPL
, class TimeDurationType = TimeDuration<STTCL_DEFAULT_TIMEDURATIONIMPL>
>
class SttclSemaphore;
```

Impl specifies the implementation class.

TimeDurationType specifies the time duration representation class to use.

Required implementation interface:

```
Impl(unsigned int initialCount);
```

Constructor for implementation class.

```
void wait();
```

Waits blocking forever until the semaphore is incremented.

```
bool try_wait(const TimeDurationType& timeout);
```

Waits until the semaphore is incremented within the specified timeout duration.

void post();
Increments the semaphore.

2.4.4 *sttcl::TimeDuration<>*

The **TimeDuration** class declares an abstraction for a time duration representation implementation class.

Template signature:

```
template<class Implementation = STTCL_DEFAULT_TIMEDURATIONIMPL>  
class TimeDuration;
```

Implementation specifies the implementation class.

Type definitions:

```
typedef typename Implementation::NativeTimeDuration NativeTimeDuration;
```

Required implementation interface:

```
Implementation(unsigned int argHours, unsigned int argMinutes, unsigned int  
argSeconds, unsigned int argMilliseconds, unsigned long argMicroSeconds, unsigned  
long argNanoSeconds);
```

Constructor for implementation class.

```
Implementation(const Implementation& rhs);
```

Constructor for implementation class.

```
Implementation& operator=(const Implementation& rhs)
```

Assignment operator for class TimeDuration.

```
bool operator==(const Implementation& rhs) const
```

Equality comparison operator for implementation class.

```
bool operator!=(const Implementation& rhs) const;
```

Inequality comparison operator for implementation class.

```
bool operator<(const Implementation& rhs) const
```

Less comparison operator for implementation class.

```
bool operator<=(const Implementation& rhs) const
```

Less or equality comparison operator for implementation class.

```
bool operator>(const Implementation& rhs) const
```

Greater comparison operator for implementation class.

```
bool operator>=(const Implementation& rhs) const
```

Greater or equality comparison operator for implementation class.

```
Implementation& operator+=(const Implementation& rhs)
```

Adds the rhs time duration to this instance.

```
Implementation& operator-=(const Implementation& rhs)
```

Subtracts the rhs time duration from this instance.

```
Implementation& operator*=(int factor)
```

Multiplies the time duration from of this instance with factor.

```
Implementation& operator/=(int divider)
```

Divides the time duration from of this instance by divider.

```
int hours() const
```

Gets the hours represented in this instance.

```
int minutes() const
```

Gets the minutes represented in this instance.

int seconds() const
Gets the seconds represented in this instance.

int milliseconds() const
Gets the milliseconds represented in this instance.

int microseconds() const
Gets the microseconds represented in this instance.

int nanoseconds() const
Gets the nanoseconds represented in this instance.

void hours(int newHours)
Sets the hours represented in this instance.

void minutes(int newMinutes)
Sets the minutes represented in this instance.

void seconds(int newSeconds)
Sets the seconds represented in this instance.

void milliseconds(int newMilliSeconds)
Sets the milliseconds represented in this instance.

void microseconds(int newMicroSeconds)
Sets the microseconds represented in this instance.

void nanoseconds(int newNanoSeconds)
Sets the nanoseconds represented in this instance.

const NativeTimeDuration& getNativeValue() const
Gets the native time duration representation.

2.4.5 sttcl::internal::SttclEventQueue<>

The **SttclEventQueue** class represents an abstraction for a thread safe queue to communicate events between threads asynchronously. Currently the **SttclEventQueue** class is used to dispatch event calls to the **State<>** class instances contained in a **Region<>** to be executed in a separate thread.

Template signature:

```
template
< class T
, class TimeDurationType = TimeDuration<STTCL_DEFAULT_TIMEDURATIONIMPL>
, class SemaphoreType = SttclSemaphore<STTCL_DEFAULT_SEMAPHOREIMPL, TimeDurationType>
, class MutexType = SttclMutex<STTCL_DEFAULT_MUTEXIMPL, TimeDurationType>
, class InnerQueueType = STTCL_DEFAULT_DEQUEIMPL(T)
>
class SttclEventQueue;
```

3 STTCL Demo applications

3.1 Demo1

The demo1 Application shows how to setup a simple state machine using STTCL.

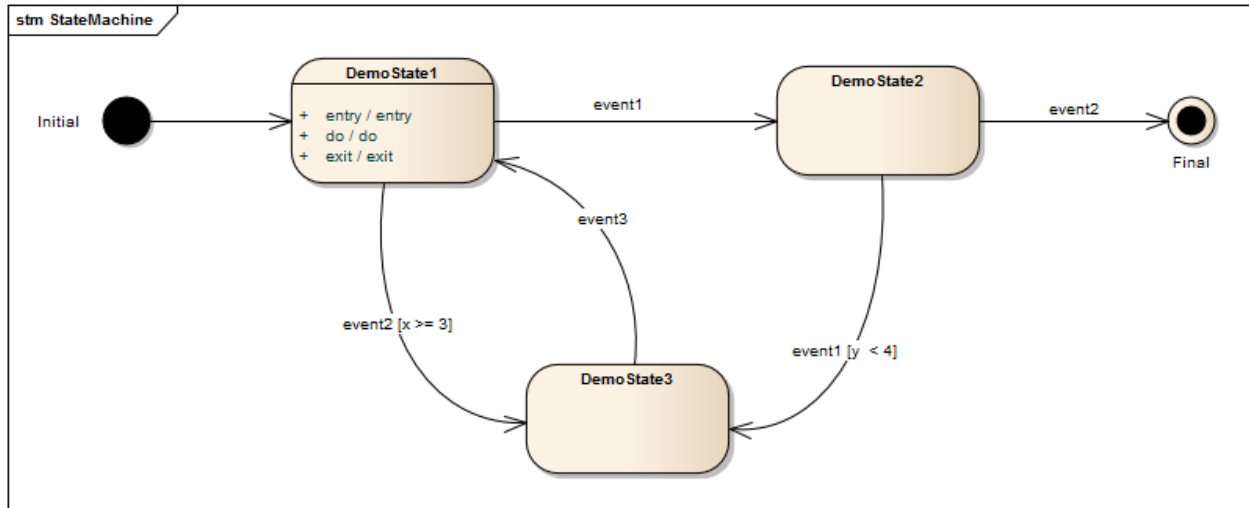


Fig 11: Demo1 State diagram

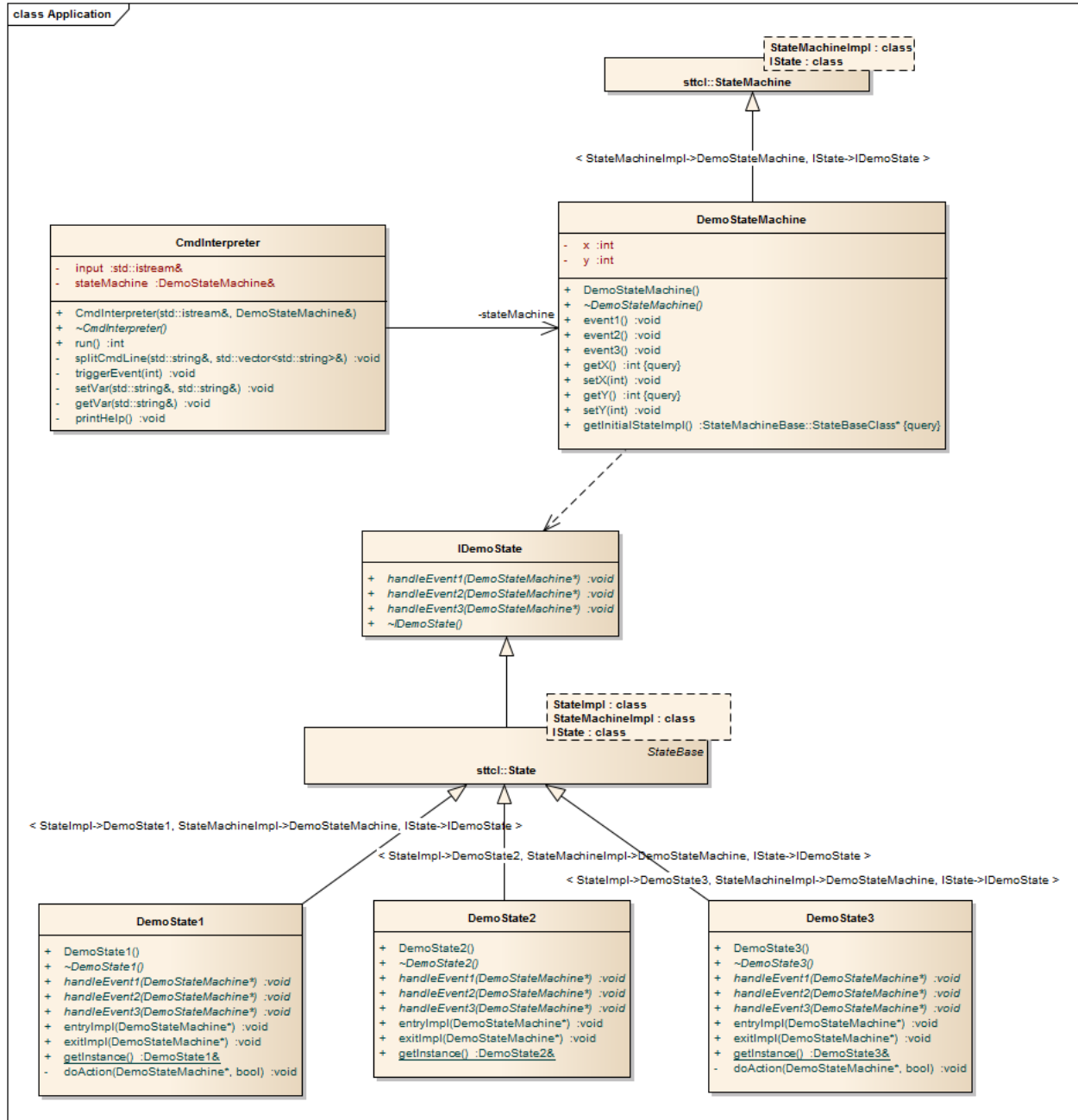


Fig 12: Demo1 Application Class diagram

3.2 Demo2

The demo2 Application shows using an ActiveState in the same state machine as used in demo1.

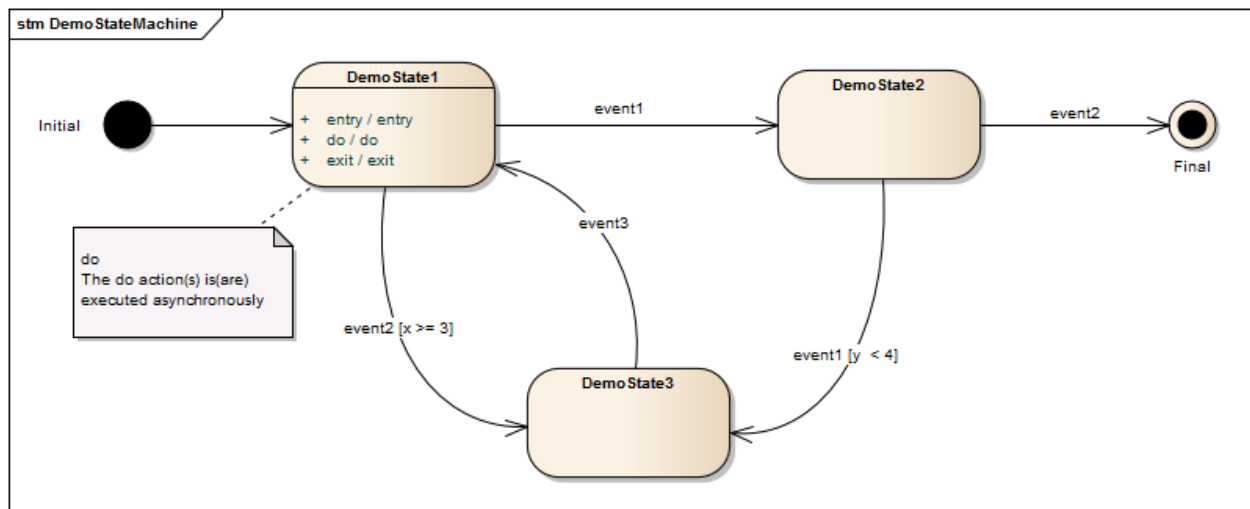


Fig 13: Demo2 State diagram

3.3 Demo3

The demo3 Application shows how to build composite states using STTCL.

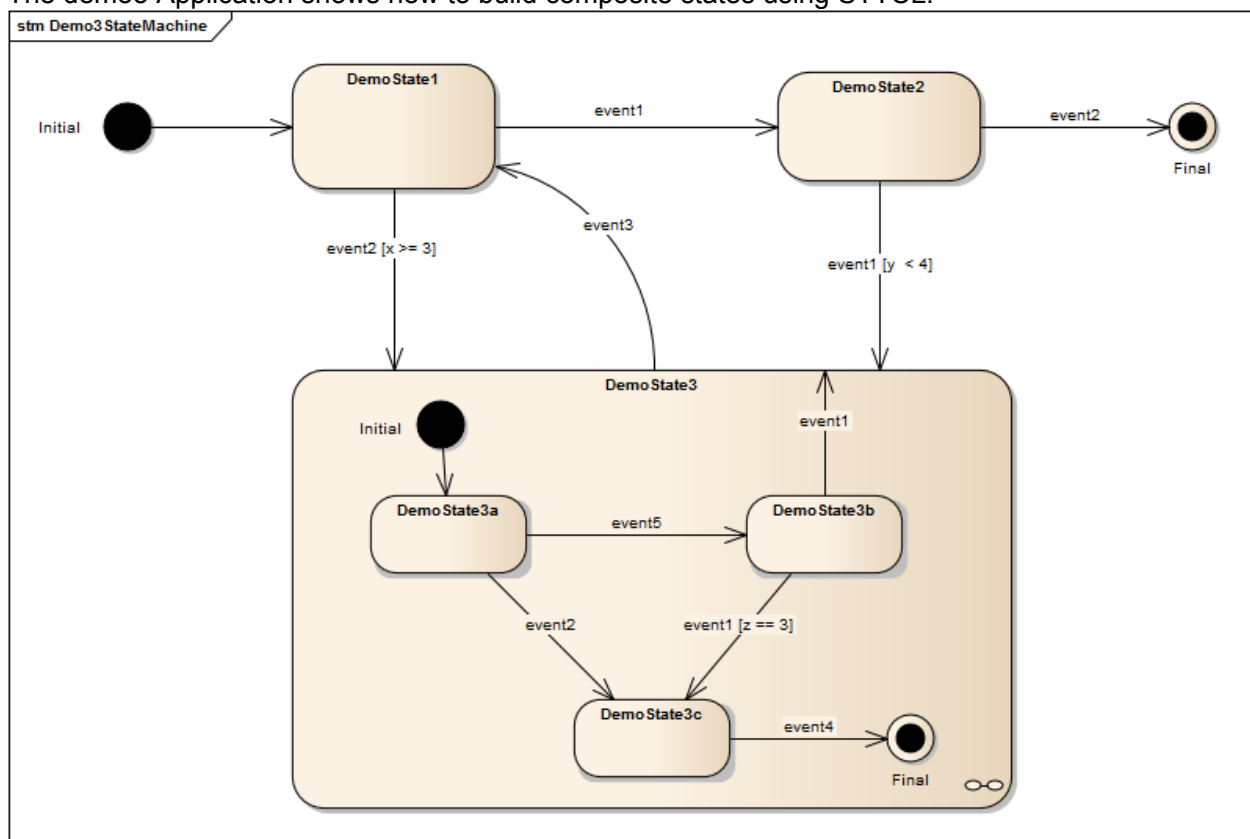


Fig 14: Demo3 State diagram

3.4 Demo3a

The demo3a Application shows how to apply a shallow state history in a composite states using STTCL.

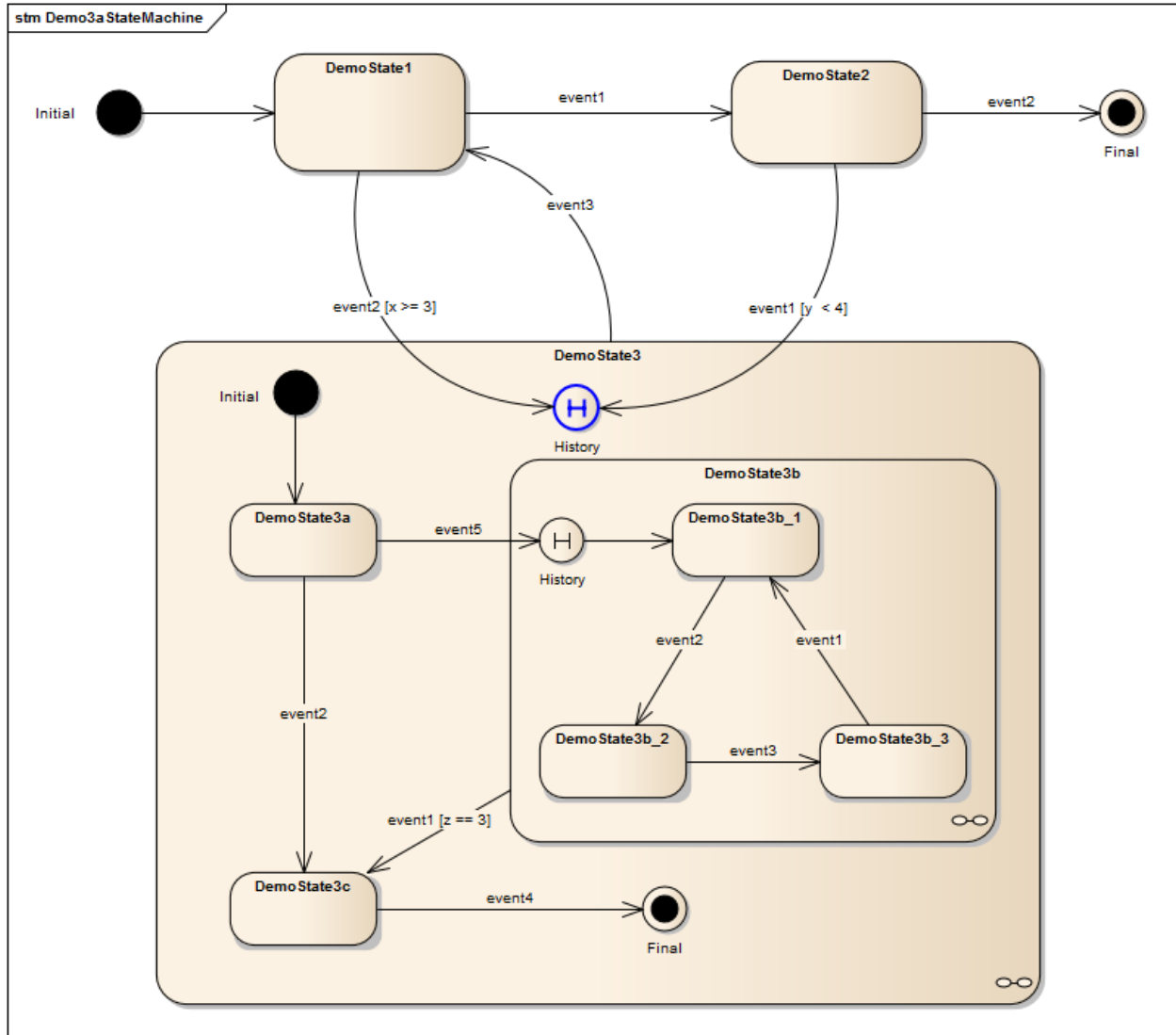


Fig 15: Demo3a State diagram

3.5 Demo3b

The demo3b Application shows how to apply a deep state history in a composite states using STTCL.

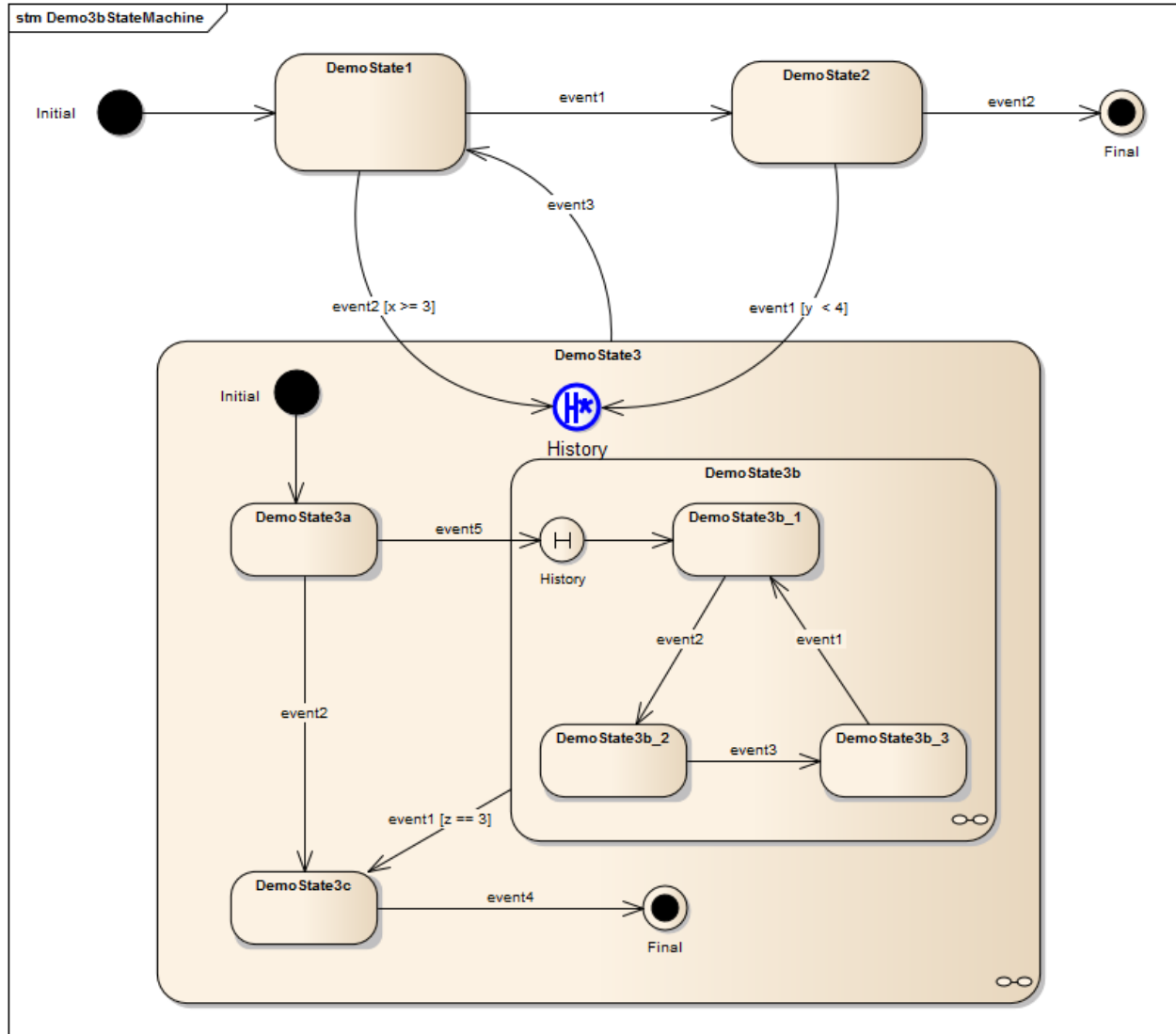


Fig 16: Demo3b State diagram

3.6 Demo4

The demo4 Application shows how to setup a concurrent composite state using STTCL.

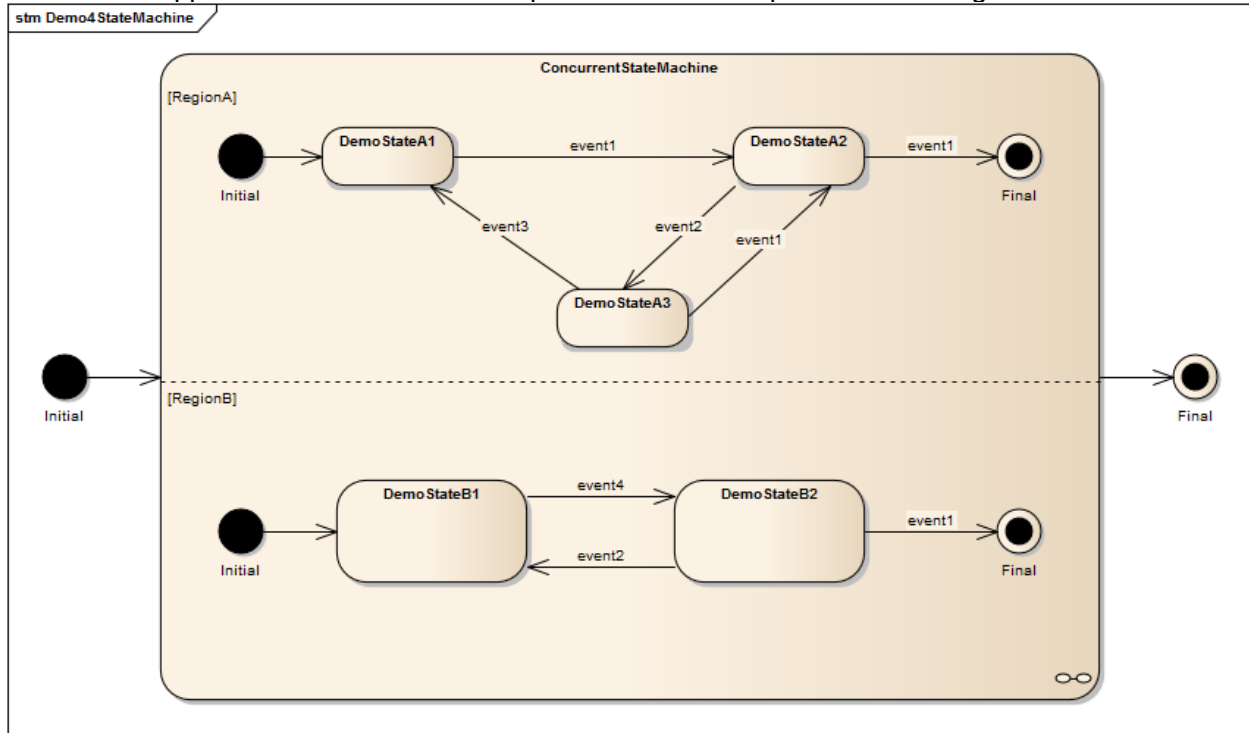


Fig 17: Demo4 State diagram

3.7 Demo4a

The demo4a is basically the same as Demo4, but demonstrates how you can pass event arguments with the event calls to the orthogonal regions.

The Demo4 and Demo4a have the problem, that the detection of the sub statemachine completion is deferred until the main state machine receives another event. Have a look at Demo6 how to solve this, by refactoring the main state machine to receive any events,- outer or inner -, asynchronously.

3.8 Demo5

The demo5 Application shows how direct transitions can be handled using STTCL.

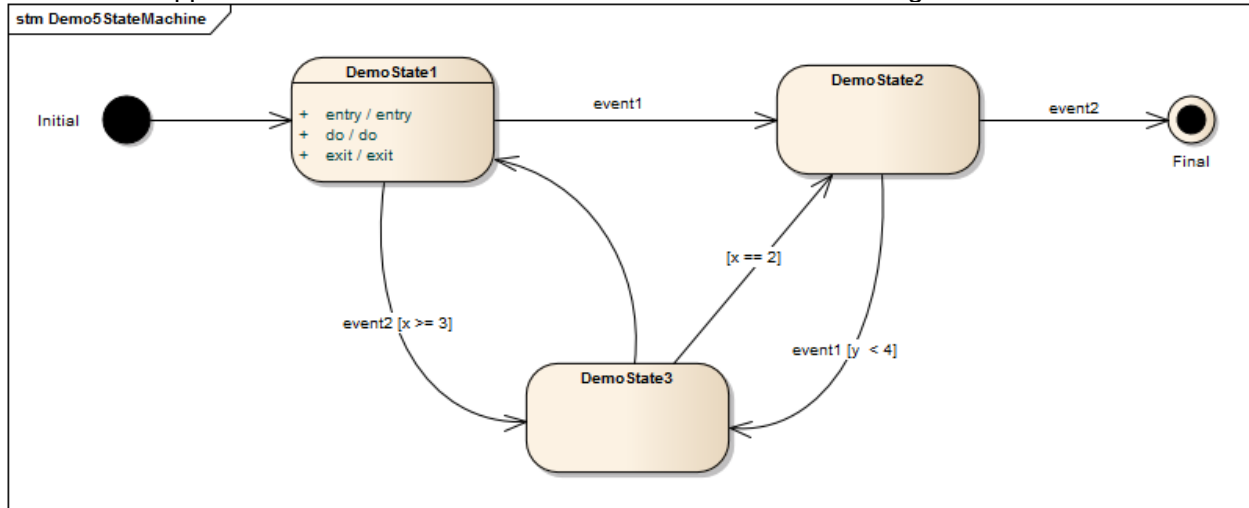


Fig 18: Demo5 State diagram

3.9 Demo5a

The demo5a Application shows how direct transitions can be guarded asynchronously.

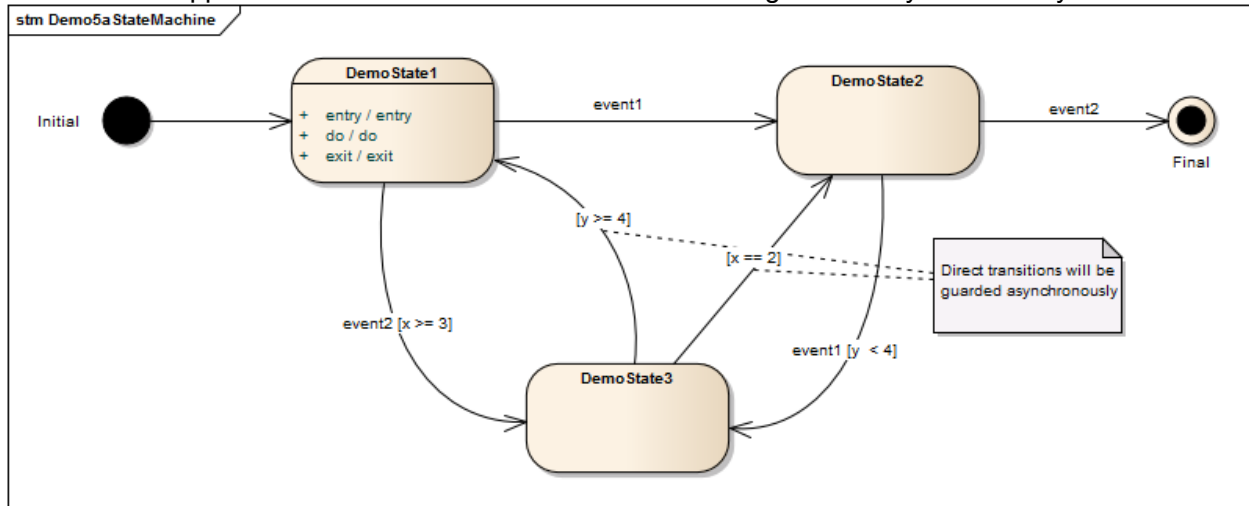


Fig 19: Demo5a State diagram

3.10 Demo6

The demo6 Application shows how to setup a concurrent composite state using STTCL using an asynchronous event signaling design for the main state machine. The state machine is the same as in Demo4 and Demo4a, but the main state machine class `DemoStateMachine` runs its own internal

thread to receive events either from input, or raised from internal (composite) states.