C++ State Template Class Library STTCL Concept

Version 2.0 / Feb 11, 2012

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

Table of Contents

History:

Version	Date	Changes	Author
2.0	11.02.12	Initial version	G. Makulik

1 Overview

The C++ **St**ate **T**emplate **C**lass **L**ibrary 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

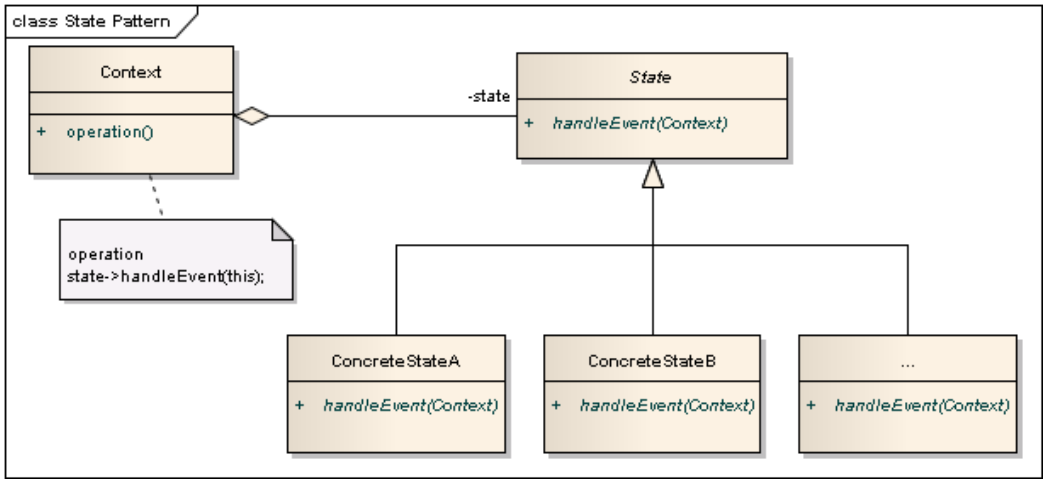


*Fig 1: The structural UML representation of the GoF **State** design pattern*

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.

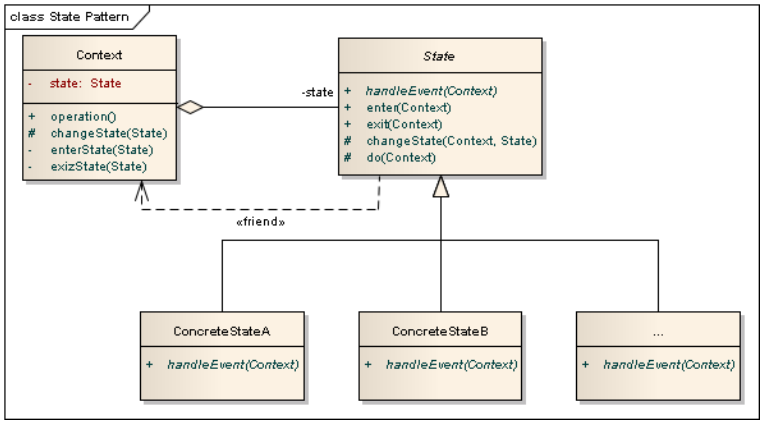


Fig 2: GoF State design patten implementation details

Fig 2 Shows a refined *UML Class Diagram* of the GoF ***State*** design pattern to depict a proposal for some 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.

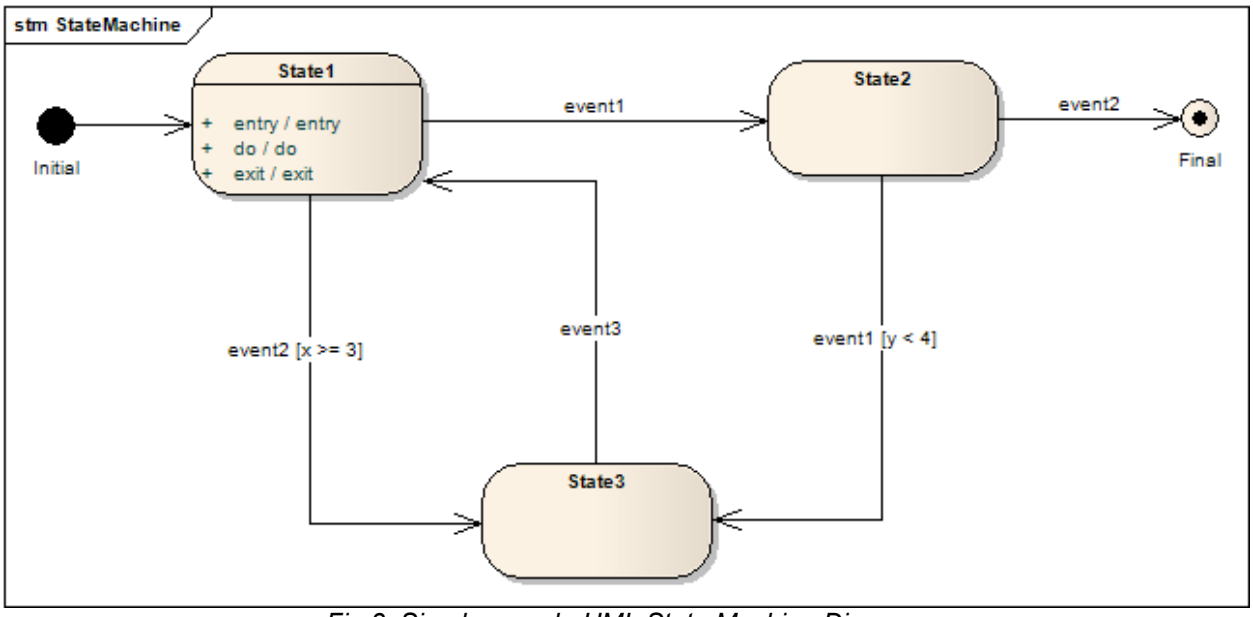
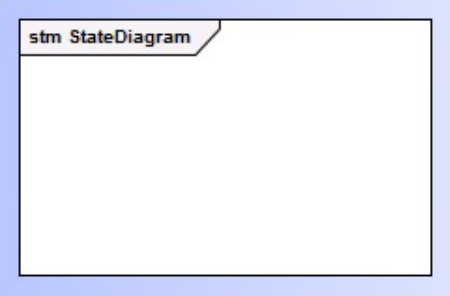
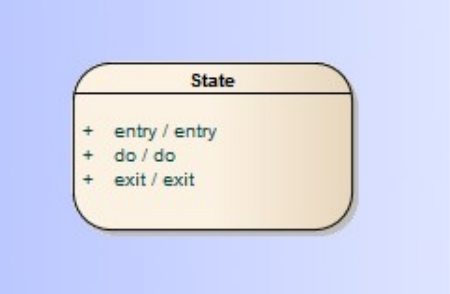
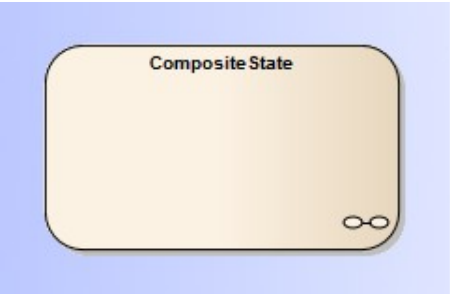
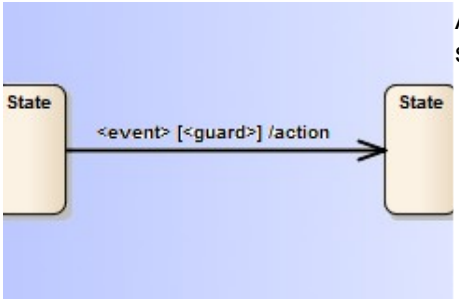


Fig 3: Simple sample UML State Machine Diagram

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

UML state machine diagram notation	GoF *State* design pattern element
 A transition between two states	A call of the **`Context::changeState()`** operation.
<event> An event that triggers the associated transition	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.
[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.	A conditional statement inside the **`Context::operation()`** or ***`State::handle()`*** operations, that decides to call **`Context::changeState()`**.
: action, … A list of specified event triggered operations	The specified action operations are called inside the implementation ***`State::handleEvent()`*** event handler operation[1]. The calling order may be unspecified.

[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 additionl behaviors can be implemented before calling the **`sttcl::State<>::changeState()`** operation.

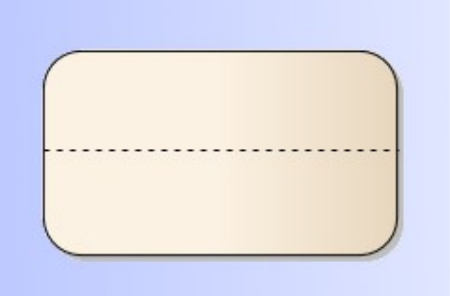
UML state machine diagram notation	GoF *State* design pattern element
 Separates concurrently active regions within a composite state or state machine diagram.	This requires concurrent program execution mechanisms (e.g. threading) supported by an operating system. Each region defines an associated ***`State`*** class reference, to delegate the event handling to a ***`State::handleEvent()`*** operation concurrently. A single region can also be considered as a concrete composite state implementation, that supports a non blocking ***`State::do()`*** 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 **`ConcurrentCompositeState<>`** and **`Region<>`** template base classes to design these UML features.

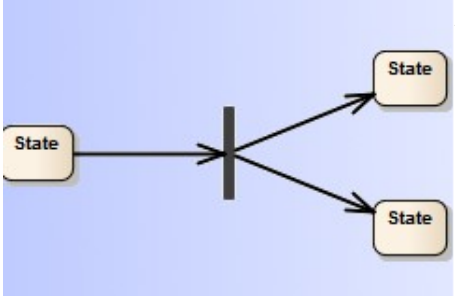
Table 1: Basic UML State Machine Elements

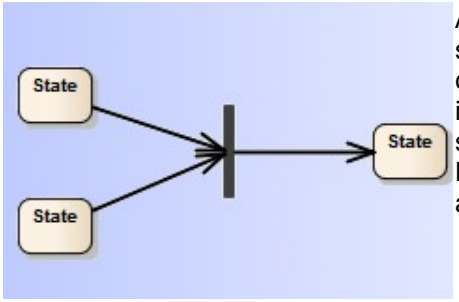
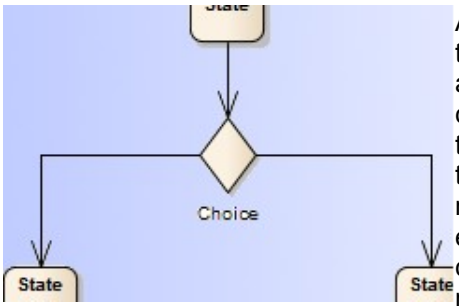
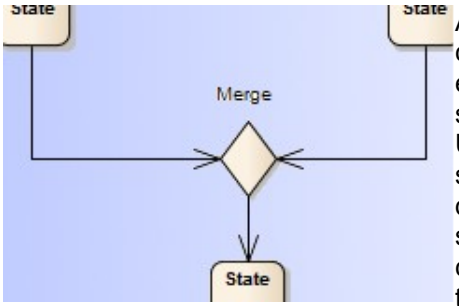
There's a number of advanced *UML State Machine Diagram* notation elements, that do not directly map to any conceptual element described in the GoF ***State*** design pattern. These can be mapped to certain aspects of implementation and behavior though.

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 *State* design pattern implementation
Pseudo-States		
	An entry point of the state machine diagram. Initial pseudo-states never have the target role of a transition.	A constructor call to create a **`Context`** class instance in the simplest case. The constructor calls the **`Context::changeState()`** operation to set the initial ***`State`*** reference. 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 ***`State`*** reference to set, to the associated **`Context::operation()`** event operations. This behavior may be encapsulated in a **`Context::initialize()`** operation.
	An exit point of the state machine diagram. Final pseudo-states never have the source role of a transition.	A ***`State`*** class implementation, that never calls the **`Context::changeState()`** operation.
	An exit point of a state machine or composite state triggered by the incoming transition's event.	A **`Context::finalize()`** operation, that calls the actual ***`State`*** references **`exit()`** operation.
	Exits the composite state or state machine triggered by the incoming transition's event.	A destructor call to a **`Context`** class instance in the simplest case.

UML state machine diagram notation	GoF *State* design pattern implementation
 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.	The composite `Context` class instance must keep track of the most recent sub ***`State`*** reference, when the composite ***`State`*** classes `exit()` operation is called. When the composite ***`State`*** classes `enter()` operation is called later on, the composite `Context` class directly transits to the remembered most recent sub ***`State`*** reference. STTCL composite state classes provide the ***`HistoryType`*** template parameter to determine the history behavior.
 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.	The same behavior as described for the deep History. But In case, that a reentered sub ***`State`*** reference also represents a composite state, it's composite `Context` class must be (re-)initialized. STTCL composite state classes provide the ***`HistoryType`*** template parameter to determine the history behavior.
 A Fork. Serves to split a single incoming transition into concurrently executed outgoing transitions. No guards are allowed on any associated transitions.	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.

UML state machine diagram notation	GoF *State* design pattern implementation
 A Join. Serves to synchronize multiple concurrently executed incoming transitions into a single outgoing transitions. No guards are allowed on any associated transitions.	A join represents a synchronization point (i.e. semaphore, mutex) for formerly initiated concurrently executed operations of a composite **`Context`** class. This can be implemented in a blocking operation, that waits on completion of all the associated concurrently executed operations.
 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.	A Choice can be implemented as a **`if ...else if ..else`** or **`switch`** conditional block in a ***`State`*** classes implementation (**`ConcreteStateA`**, **`ConcreteStateB`**)., that choose the appropriate target ***`State`*** reference parameter for a call to the **`Context::changeState()`** operation. **Note:** Choices don't serve to split transition paths into concurrently executed operations!
 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.	A Merge can be implemented as an operation, that is shared by a number of ***`State`*** class implementations (**`ConcreteStateA`**, **`ConcreteStateB`**), and ends up in a single call of **`Context::changeState()`** operation. The decision to call this operation is done in the implementation of the ***`State::handleEvent()`*** operation, according the associated guard condition.

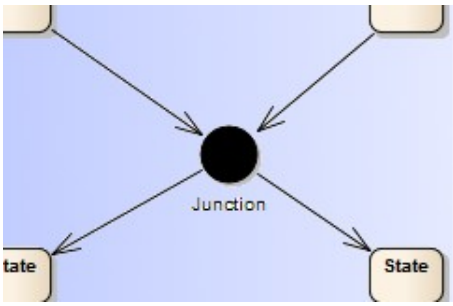
UML state machine diagram notation	GoF *State* design pattern implementation
 A Junction. Serves to share transition paths for the incoming transitions. The incoming and or outgoing transitions have guard conditions associated. Incoming transitions are shared between the source states. Outgoing transitions 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.	The Junction serves complex conditional path transitions that can be implemented in a similar way as the Choice and Merge pseudo-states. 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 `State` class. Such implementation should not affect the `Context` classes attributes, but just serve to forward incoming events to outgoing transitions (i.e. `Context::changeState()` calls).

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.

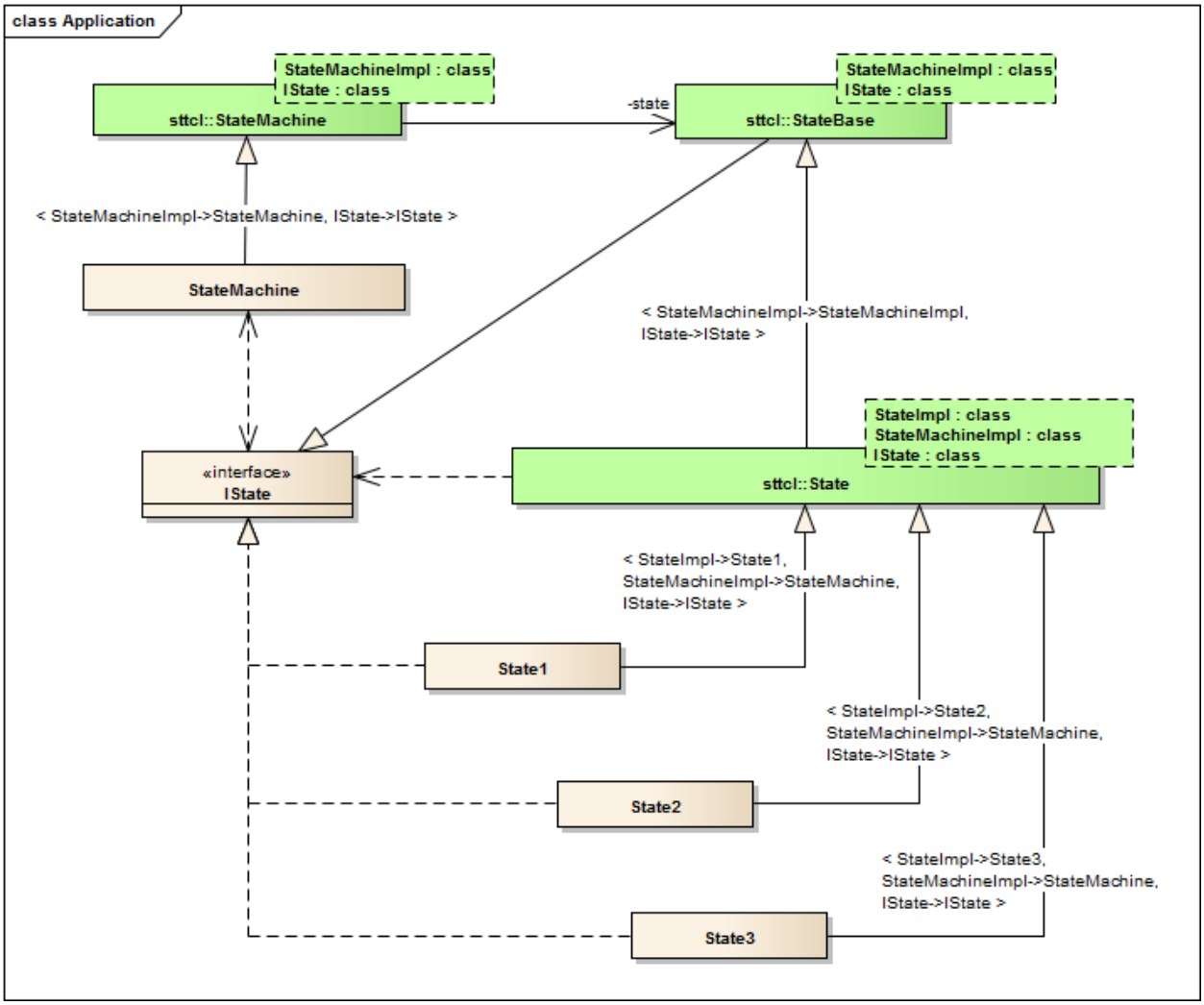


*Fig 4: The basic STTCL **State** design pattern abstraction*

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

2.1.2 Concurrency aspect

2.1.3 Direct transition aspect

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

2.1.4 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

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

2.1.5 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_MUTEXDIMPL 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.

2.2 STTCL Base Classes

The STTCL template base classes provide default impleentations 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 behaviour and the specific method is called using a static cast to the implementation class.

2.2.1 sttcl::StateMachine<>

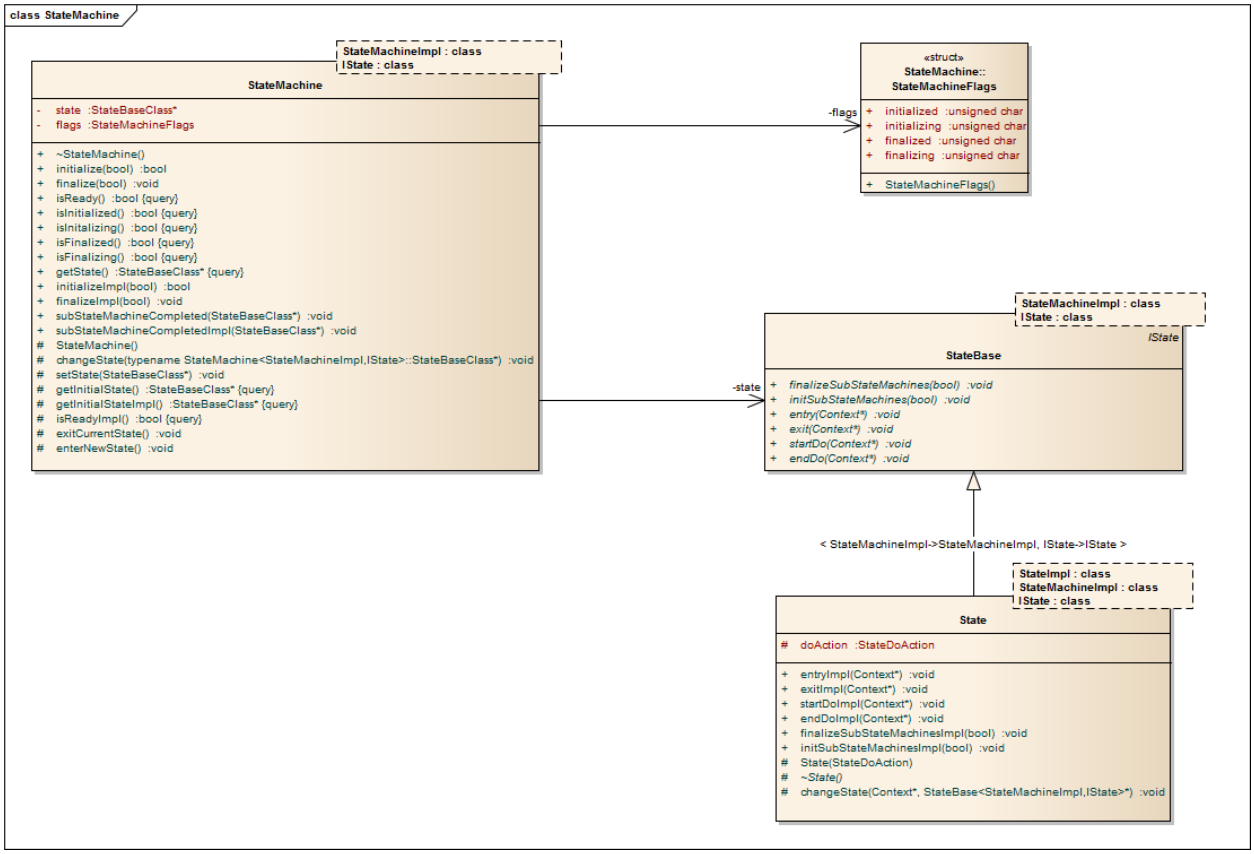


Fig 5: sttcl::StateMachine<> class diagram

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.

2.2.2 sttcl::State<>

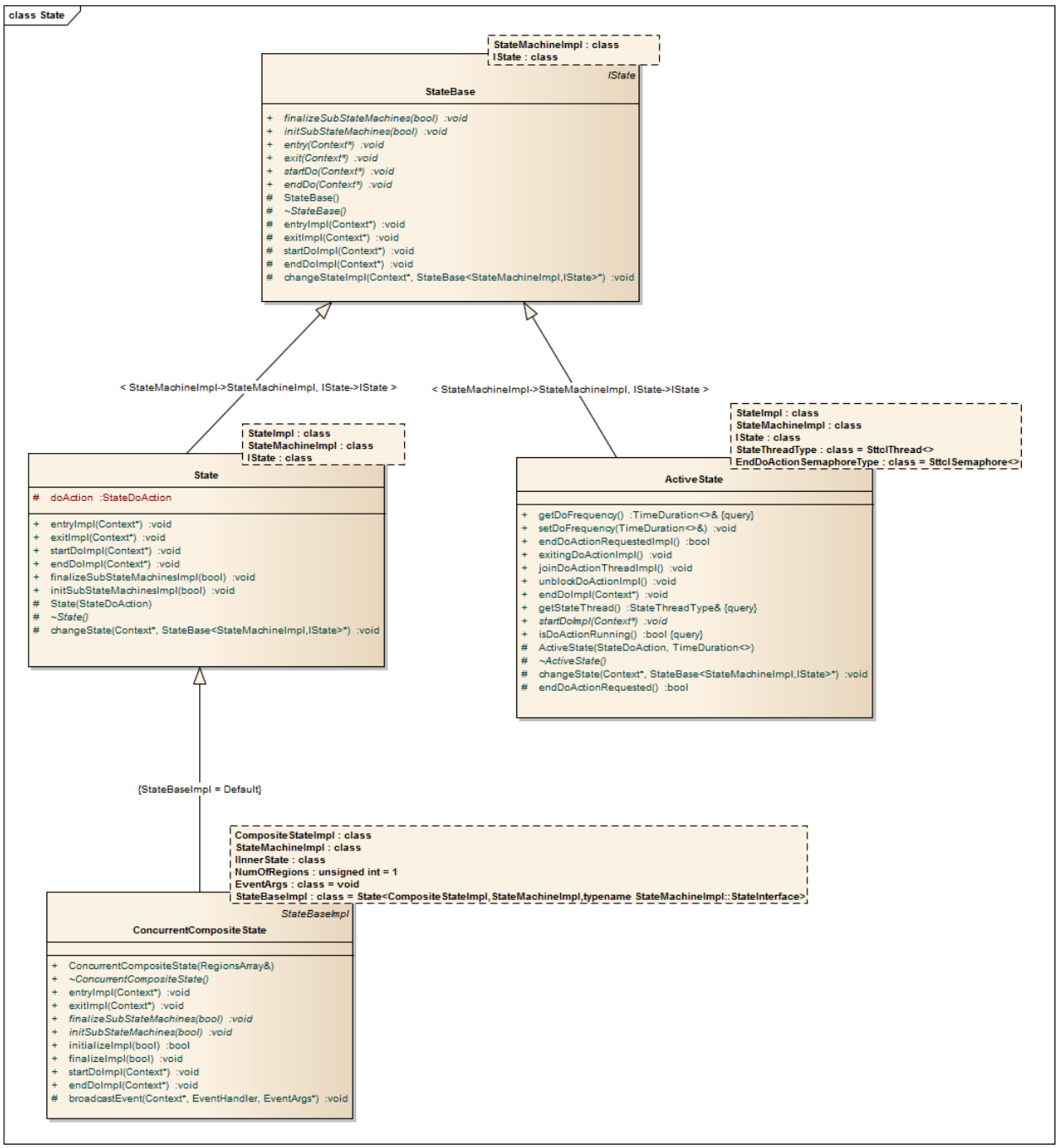


Fig 6: sttcl::State<> class diagram

Template signature:

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

`StateImpl` specifies the inheriting class.
`StateMachineImpl` specifies the containing state machine implemetation.
`IState` specifies the internal state interface class.

The `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()`

`startDoImpl()`

`endDoImpl()`

`exitImpl()`

`initSubStateMachinesImpl()`

`finalizeSubStateMachinesImpl()`

`checkDirectTransitionImpl()`

2.2.3 sttcl::ActiveState<>

2.2.4 sttcl::CompositeState<>

2.2.5 sttcl::ConcurrentCompositeState<>

2.2.6 sttcl::Region<>

2.3 STTCL configuration adapters

2.3.1 sttcl::SttclThread<>

2.3.2 sttcl::SttclMutex<>

2.3.3 sttcl::SttclSemaphore<>

2.3.4 sttcl::TimeDuration<>

3 STTCL Demo applications

3.1 Demo1

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

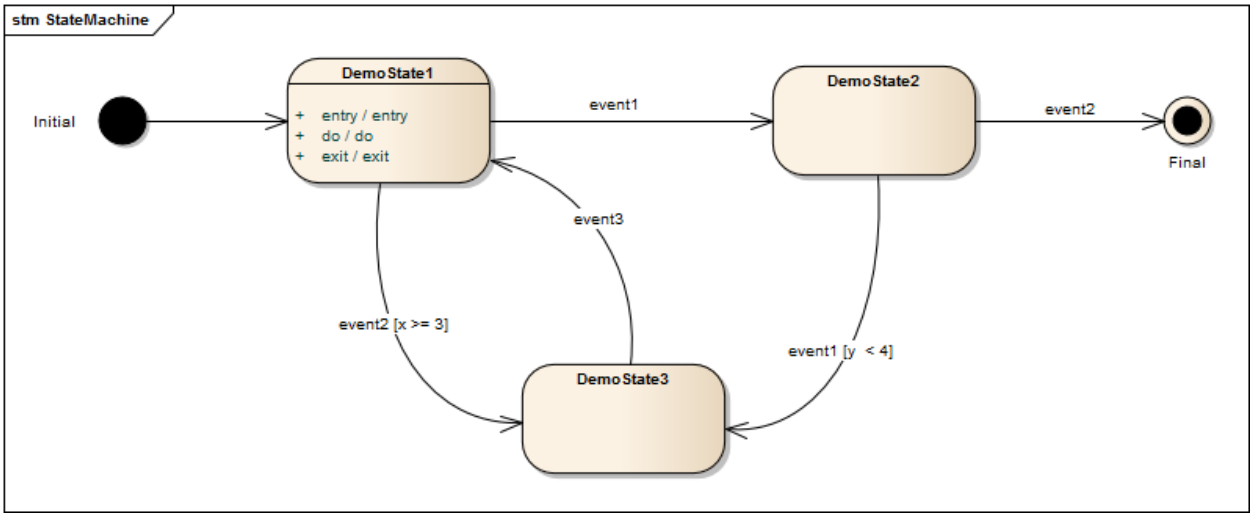


Fig 7: Demo1 State diagram

3.2 Demo2

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

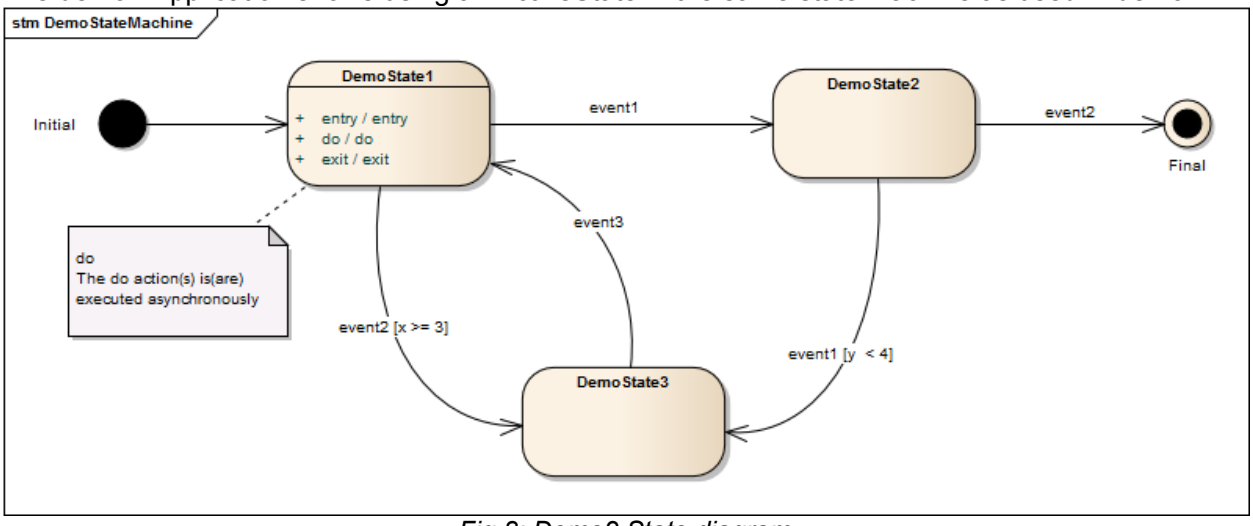


Fig 8: Demo2 State diagram

3.3 Demo3

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

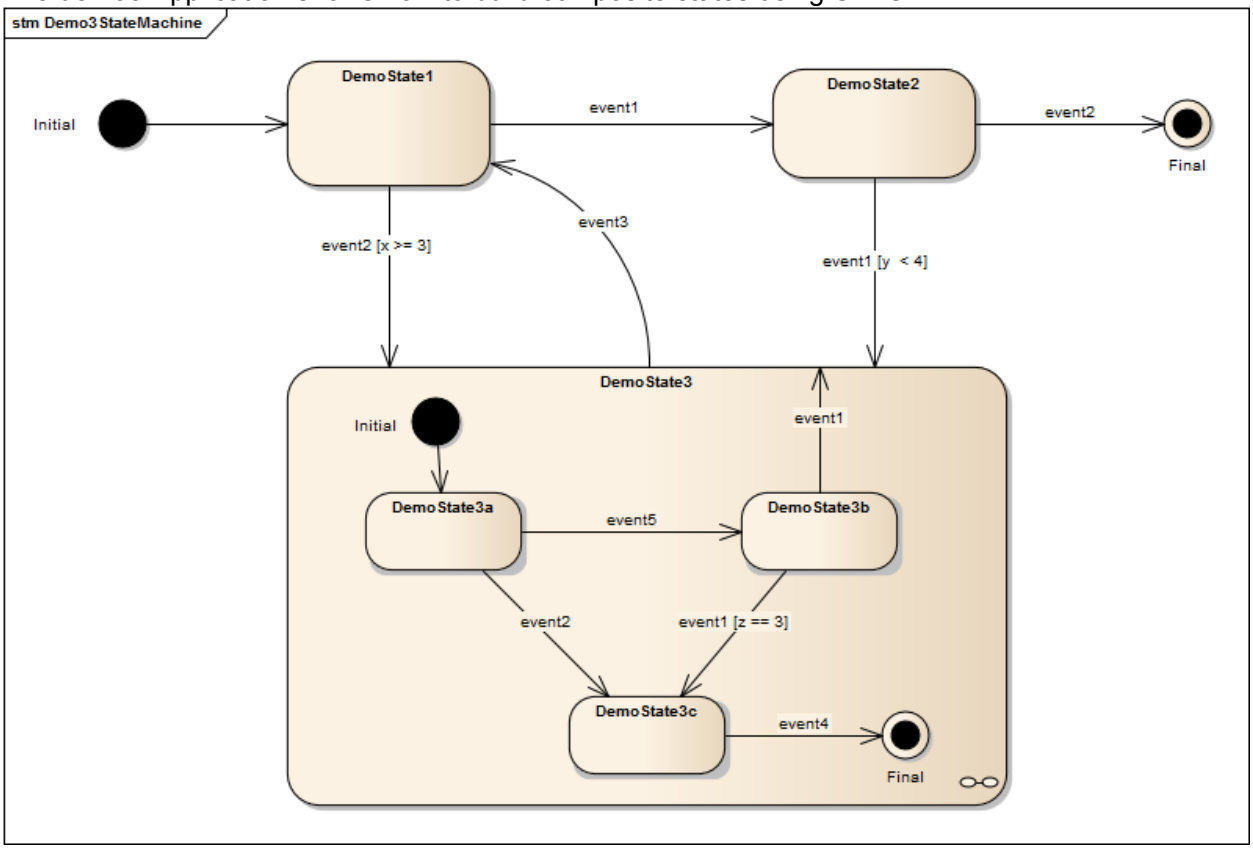


Fig 9: Demo3 State diagram

3.4 Demo3a

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

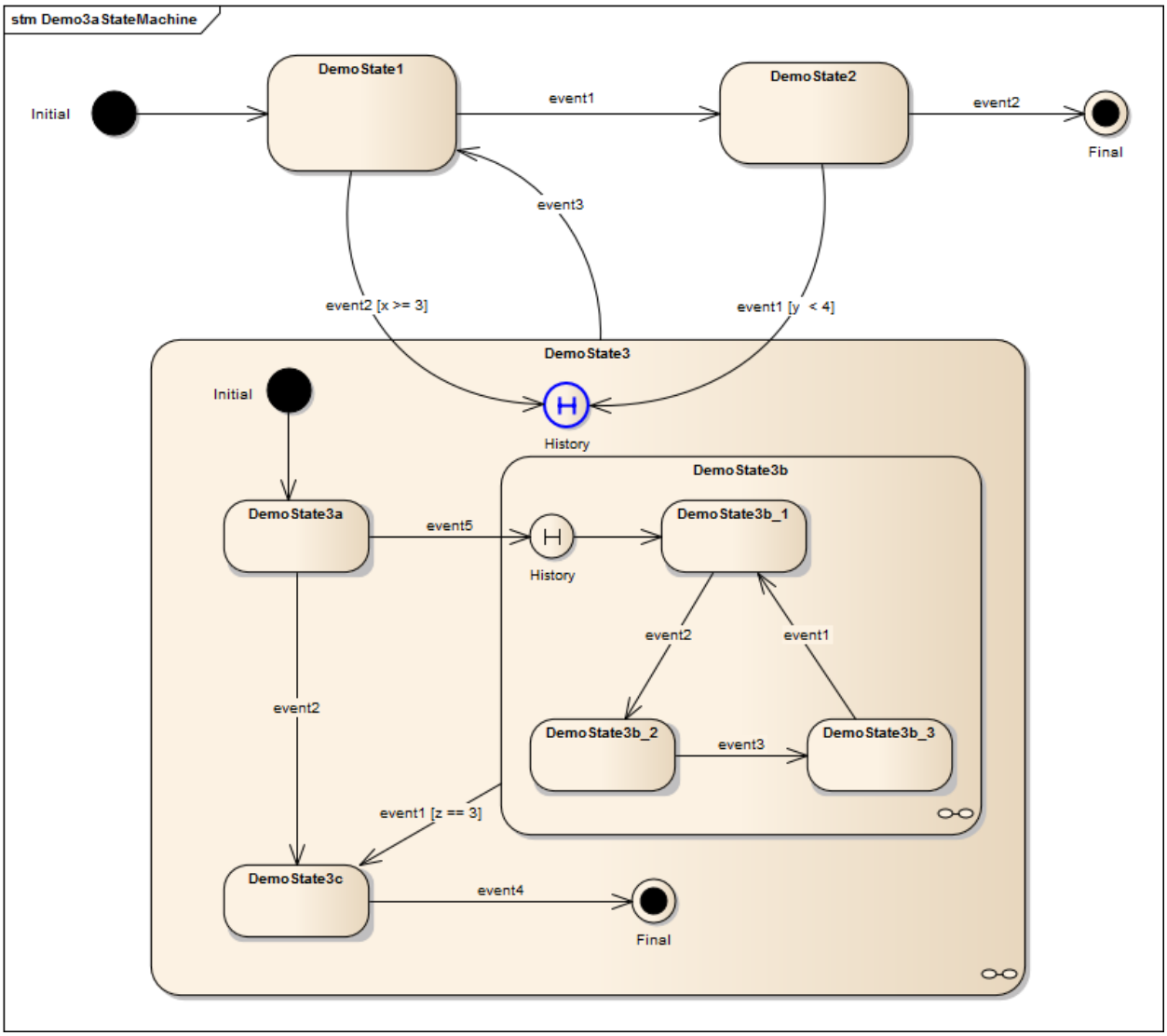


Fig 10: Demo3a State diagram

3.5 Demo3b

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

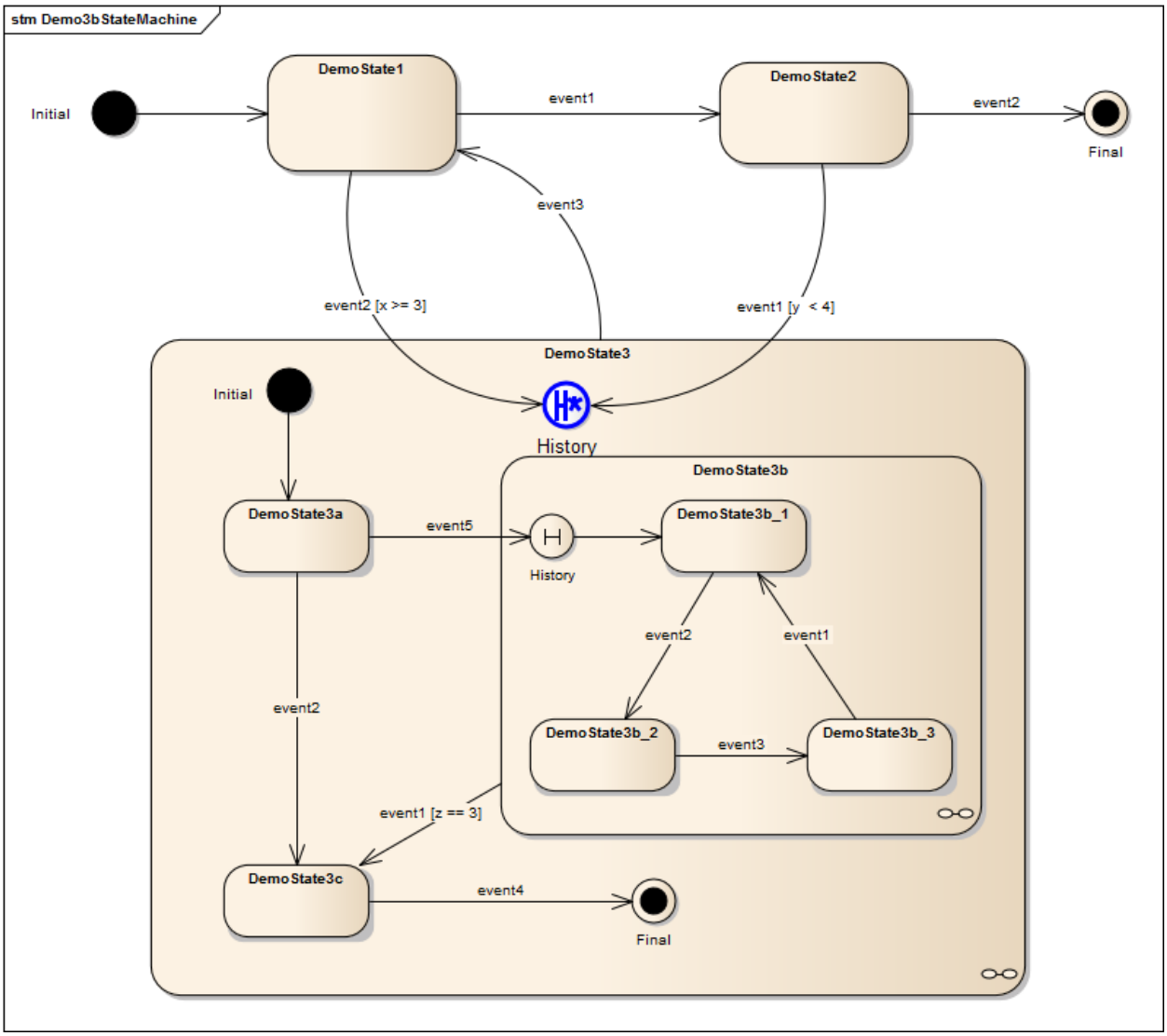


Fig 11: Demo3b State diagram

3.6 Demo4

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

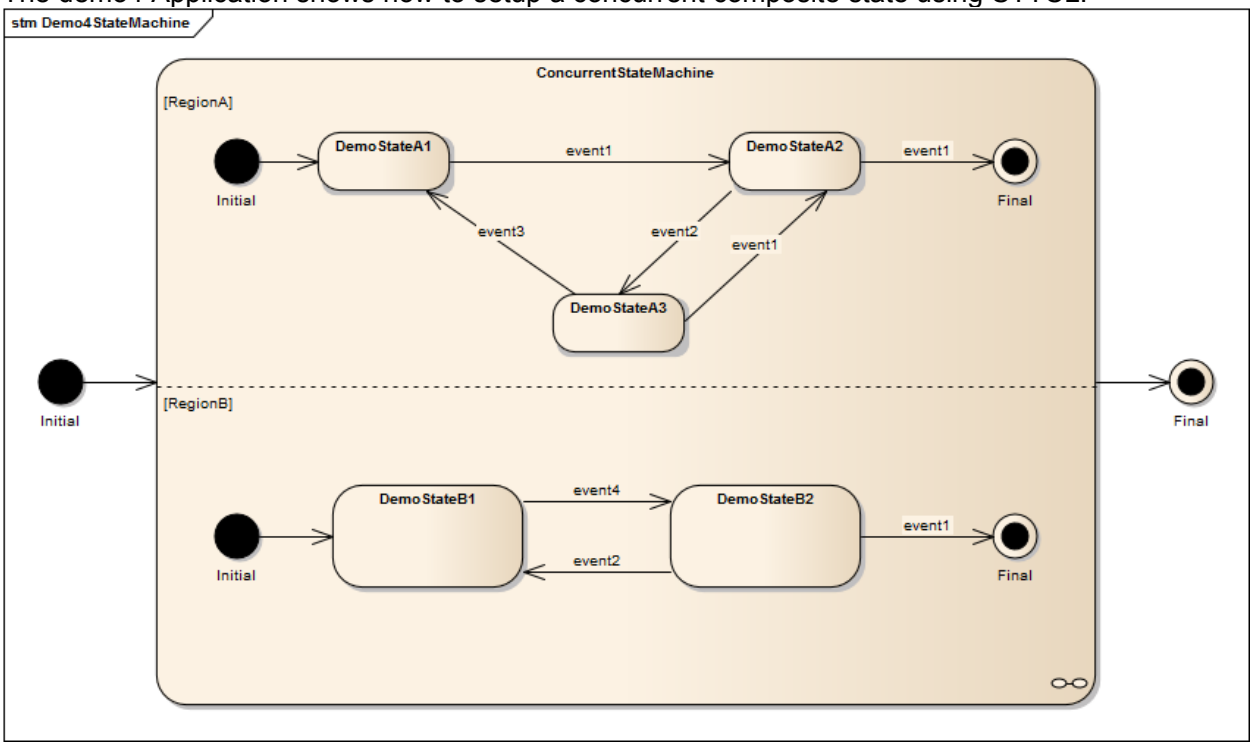


Fig 12: Demo4 State diagram

3.7 Demo5

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

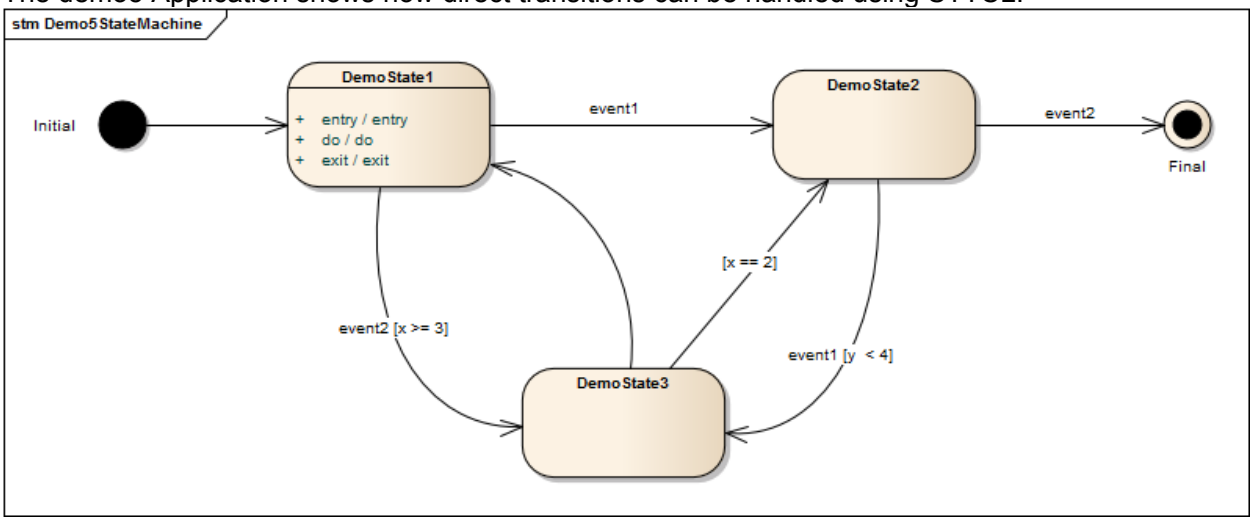


Fig 13: Demo5 State diagram

3.8 Demo5a

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

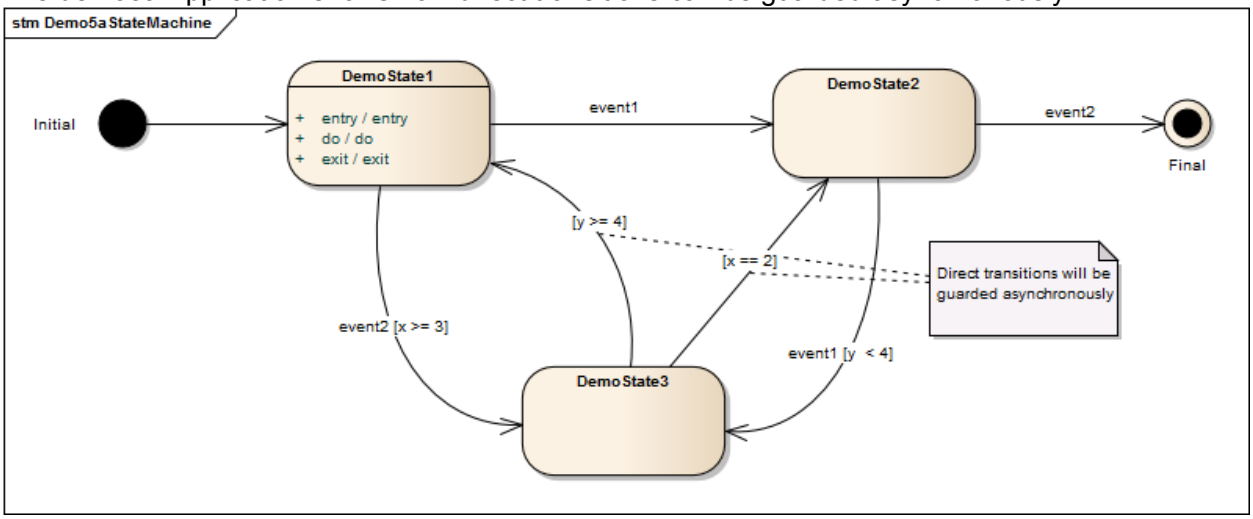


Fig 14: Demo5a State diagram