SOL Software d.o.o.
www.sol.rs

Public

SOLoist State Machines

Semantics and Usage

Project: SOLoist V4
Document Type: Project Documentation (PD)
Document Version: 1.0
Date: 01.06.2015

1. Table of Contents

Table of Contents

2. Introduction

This document describes the semantics, usage, and principles of implementation of UML state machines in SOLoist V4 (www.soloist4uml.com).

The semantics of state machines (SM) follows the profiling of standard UML state machines as described in the book:

> [1] Milicev D., *Model-Driven Development with Executable UML*, Wiley/Wrox, July 2009, ISBN 9780470481639

with some minor adaptations as described in this document.

The usage in modeling is described for the StarUML modeling tool that is currently used in SOLoist 4. As this tool does not support all the concepts of SMs, SOLoist takes some workarounds in their modeling, different from the standard. These modeling workarounds are also described in this document.

3. Overview of State Machines in SOLoist 4

An SM is normally modeled as a behavioral feature attached to a class called the *host* class. The SM then describes the lifecycle of objects of that host class. At runtime, each object of the host class, called *host object*, can change its current state according to the definition of the SM.

If an SM is not associated with a class in the model, and submachines are typically stand-alone modeling elements, the host class of the SM or submachine is specified through the `HostClass` tagged value available in the SOLoist profile.

The behavior specified with an SM is ensured by the Java code generated for the SM. The code is a Java class, hereafter referred to as the *SM class*, without any non-static data fields. That is, the class generated for the SM is stateless, while the state has to be embodied in the host class (object). The relationship between the host class and the SM class is as in the Strategy design pattern (with a stateless Strategy), where the SM class plays the role of the Strategy. In order to implement the necessary behavior, methods of the SM class call back methods of the host class to:

- read the current state of the host object,
- (indirectly) execute guard conditions and actions on transitions,
- change the current state of the host object,
- perform other auxiliary processing, such as optional locking, logging, setting the timeout, etc.

In other words, the SM class provides only the control flow of the execution of the transitions and the associated activities according to the definition of the SM, while the state and all other specific functionality is hosted in the host class.

The SM class is stateless and thus its instance can be shared between all host objects. This is why this class has a single static instance (Singleton design pattern), accessible through its static `Instance()` operation.

To provide the necessary features to the SM class, the host class has to implement a certain set of operations that will be described in a separate chapter.

In terms of runtime semantics, there are two basic categories of *nodes*:

1. Steady states: These are primitive states in the SM model (except the final state) that have no substates and no enabled triggerless outgoing transitions (i.e., outgoing transitions without triggers), or a final state of a composite state, submachine, or the SM itself, whose parent composite state does not have enabled triggerless outgoing transitions. The host object resides in such a state until an explicit external trigger event occurs and a transition is fired.

2. Transient nodes: These are nodes in an SM such as composite states, pseudostates (choice points and initial pseudostates), entry and exit points, primitive states with enabled triggerless outgoing transitions, and a final state within a composite state with an enabled triggerless outgoing transition. The host object never resides in such a transient node. Instead, these nodes, along with their guards and effects, constitute an integral part of compound transitions: when a transition is fired, all passing nodes along the taken transition path are passed through and the actions associated with them are executed, until a steady state is reached, in a single run-to-completion step.

Terminate pseudostates are not supported. Any kind of concurrency in SMs, such as concurrent states and regions, forks/joins, etc., is not supported. Entrance into a composite state with history is not supported.

In the current version of SOLoist, SMs do not have any footprint in the UML reflection repository.

4. Event Processing

After its initialization, a host object normally resides in a steady state of the SM until it is triggered. The SM machine is triggered by calling the operation of an SM class instance:

```
boolean process (HostClass hostObject, String triggerName);
```

This operation accepts the host object as its first parameter and the name of the trigger (exactly the same as specified in the model) as the second parameter. The operation returns true if the steady state has been changed after the entire transition.

When a host object is being initialized, e.g., somewhere in its constructor, this operation has to be called for the host object with "`init`" as its second parameter, to fire the initial transition. The SM itself should contain exactly one topmost (normally composite) state named "`TOP`", without any incoming or outgoing transitions; the initial transition will be fired from the initial pseudostate within that topmost state.)

When this operation is called, it performs the (possibly compound) transition with the "run-to-completion" semantics [1] in the following sense: it does not return the control until a steady state or a terminate pseudostate is reached by the transition.

When called, the operation `process()` performs principally the following:

1. Calls back the operation `HostClass.lock()` of the host object. By this, the host object has the opportunity to perform explicit (pessimistic) or implicit (optimistic) locking, in order to avoid concurrency conflicts when being triggered from different threads of control. For example, the host object can simply perform a read action on its state attribute to read the version of the object and detect concurrency conflicts on the transaction commit.

2. Calls back the operation `HostClass.getState()` of the host object to read the current steady state. The host class can store the state in an attribute.

3. Provided that an outgoing transition is fired, performs the compound transition through transitions, transient nodes and composite states, until a steady state is reached. The details of the effects of this part of processing will be described later for each particular concept.

On each entrance to a state or exit from a state, composite state, or pseudostate, it calls back the operation `HostClass.logStateEntry()` and `HostClass.logStateExit()`, respectively. By this, the host object has the opportunity to log the transition path trough the SM in its internal structure. It can simply ignore this call, or store the trace in an attribute of type `File`, for example.

4. If a steady state has been reached, calls back the operation `HostClass.setState()` of the host object to store its new current steady state.

5. Calls back the operation `HostClass.setTimeout()` of the host object. By this, the host object has the opportunity to store the deadline when the SM has to be automatically triggered by a timed event (timeout). Whenever a state with a timeout is exited, it calls back the operation `HostClass.resetTimeout()` of the host object, to reset the timer for this state.

6. Calls back the operation `HostClass.unlock()` of the host object. By this, the host object has the opportunity to unlock itself, if it has been locked explicitly.

7. Returns true if the steady state has been changed and false otherwise.

Note: To ensure isolation and fault tolerance of the object space, the execution of the described process has to be enclosed in a SOLoist transaction. Managing the transaction is the responsibility of the caller's context of the `process()` operation.

5. Supported Concepts

This chapter describes particular concepts of SMs supported in SOLoist, their usage in modeling with StarUML, their semantics, and some hints about the implementation.

5.1. Expressions

For the specification of guards and effects of transitions, or entry and exit actions of states, *expressions* can be used.

An expression in this context can be any Java-syntax expression that can be evaluated in the scope of the SM class, with several extensions described below in this section. Typically, and this is recommended, an expression should be just a very simple Java expression or an operation call.

The host object can be referred to from an expression by the identifier `host`. For example:

```
host.aHostClassOperation(argumentList)
```

An expression can also call an arbitrary method of the SM class that is manually defined and coded in the separate preserved section of the SM class.

5.2. Transitions

Supported Features. SOLoist SMs support the following features of transitions:

- triggers;
- guards;
- effects.

Outgoing transitions of pseudostates must not have triggers.

States and pseudostates can have one or more outgoing transitions without triggers; if a state or pseudostate has more than one outgoing triggerless transition, these transitions typically have guards. These guards should be logically complete and disjoint, although this is not checked by the framework at any time. If these guards are logically incomplete (i.e., there are cases at runtime when none of guards results to true), the SM may behave incorrectly. If these guards are not disjoint (i.e., overlapping - there are cases at runtime when more than one guard results to true), only one (arbitrary) will be taken by the implementation, while the others will always be ignored. (The case with multiple outgoing triggerless transitions without guards is logically overlapping, as their guards are all true.)

Internal transitions are not supported (these are transitions from a state to the same state, without exiting and reentering that state).

Modeling in StarUML. StarUML directly supports transitions.

Triggers are modeled directly as *triggers* in StarUML. The SM code generator is insensitive on the kind of the trigger, but for the sake of a correct modeling style, it is recommended to select either the *call* or *time* event (for timeout transitions) as a trigger.

Guards and effects are modeled directly as *guard conditions* and *effects* respectively in StarUML.

Guards and effects have to be *expressions*.

Semantics. Let *t* be a transition, and let *S* be its source node and *D* be its destination node. Let *C* be the least common owner composite state of *S* and *D*. Then, it is said that *S* and all its surrounding states up to *C* (but excluding *C*) are *exited*, and all the states starting from *C* (but excluding *C*) down to *D* are

entered by *t*.

When the host object resides in a steady state, an outgoing transition with the trigger equal to the event occurrence and with its optional guard evaluated to true (an empty guard is assumed to be true) is *enabled*. If and only if an outgoing transition with the given trigger does not exist for a nested state, its enclosing composite state is searched for such a transition, etc. recursively outwards. The first encountered such transition(s) whose guard evaluates to true is then enabled.

One arbitrary of the (possibly several) enabled transitions is *fired*, and the compound transition starts then. First, the fired transition is processed. The compound transition then continues to processing of the target node, until the processing is complete. The continuation or completion of processing of the compound transition depends on the target node and will be described for each kind of node separately.

Processing a transition includes the following:

- Executing all exit expressions of all states exited by the transition, including their nested states, from inside outwards.
- Executing the effect expression of the transition.
- Executing all entry expressions of all states entered by the transition, from outside inwards.

Implementation. For each trigger and each steady state in the SM that can potentially react on that trigger (either directly or indirectly, through its enclosing composite states), there is a branch in a `switch` structure. The branch to be taken depends on the current steady state read from the host object via `HostClass.getState()` and the trigger.

In each branch, the compound transition processing is performed as a continuous flow of control. The processing starts with the taken (fired) outgoing transition, and continues to its target node, through other nodes and transitions until it reaches a steady state.

If a transition has a guard, the entire remaining processing of it and its successor node is enclosed in an `if` clause with that guard.

If a transition has an effect, the effect expression is executed before the processing continues to the target of the transition.

5.3. Primitive States

Supported Features. Primitive states are those SM states that have no nested substates. Final states are also a kind of primitive states in UML, but they are described in a separate section and are not treated here.

SOLoist SMs support the following features of primitive states:

- entry and exit actions of states.

Modeling in StarUML. Primitive states are modeled directly in StarUML, without any annotation or stereotype.

Entry and exit actions are modeled directly in StarUML. They have to be *expressions*. Typically, and this is recommended, a state has at most one entry and at most one exit action.

Semantics. Outgoing transitions with triggers of a primitive state can be enabled only if that state is the current steady state of the host object, i.e., at the beginning of the processing of an event occurrence and compound transition. Otherwise, i.e., if the primitive state is a transient state, visited during the processing of a compound transition, only its triggerless outgoing transitions can be enabled.

When processing a compound transition reaches a primitive state, only triggerless transitions of that state can be enabled. If any of them is enabled, the transition processing goes on with one such outgoing transition being fired and the primitive state is thus transient (i.e., abandoned as soon as its entry and exit expressions are executed, within the same run-to-completion step). Otherwise, the primitive state becomes the new steady state of the host object and the compound transition processing completes.

Whenever processing enters a primitive state, its optional entry expression is executed, and the state of the host object is set to that state via `HostClass.setState()`.

Whenever processing exits a primitive state, its optional exit expression is executed.

5.4. Initial Pseudostates

Supported Features. None.

An initial pseudostate cannot have incoming transitions. It can have arbitrarily many outgoing transitions that must not have triggers.

The SM itself should contain exactly one topmost composite state named "`TOP`" with exactly one initial pseudostate in it. This pseudostate will be taken as the starting node when the `init` event is sent to the SM class via the `process()` operation.

Modeling in StarUML. An initial pseudostate is modeled directly as a pseudostate of kind `INITIAL` in StarUML.

Semantics. When an initial pseudostate is reached by a compound transition processing, it immediately continues with one of its enabled outgoing transitions that is selected to be fired. Thus, an initial pseudostate is always transient.

5.5. Choice Pseudostates

Supported Features. A choice pseudostate may have arbitrarily many outgoing triggerless transitions with covering and disjoint guard conditions. Specifically, at most one of the outgoing transitions may have the `else` guard.

Most often, and this is recommended, a choice pseudostate has only two outgoing transitions, one with a simple boolean expression, and the other being `else`. This ensures most efficient execution of a traditional if-then-else construct, because having a guard expression on each outgoing transition (i.e., an expression on one and its negation on the other) will imply execution of each of these expressions at runtime.

Modeling in StarUML. Choice pseudostates are modeled directly in StarUML as a pseudostate of kind `DECISION`: place a "choice point" from the toolbar on the diagram, and then change its pseudostate kind in its properties panel to `DECISION`.

Semantics. When a choice pseudostate is reached by a compound transition processing, it immediately continues with one of its enabled outgoing transitions that is selected to be fired. An outgoing transition is enabled if its guard evaluates to true. The `else` guard evaluates to true if and only if none of the other guards evaluates to true.

5.6. Composite States

Supported Features. Composite states are those SM states that have nested substates. SOLoist SMs

support the following features of composite states:

- entry and exit actions of states.

Neither history nor concurrency is supported.

Each composite state must have exactly one initial pseudostate in it; otherwise, the SM is ill-formed and may behave incorrectly, although this is not reported either at code generation or runtime.

Modeling in StarUML. Composite states are modeled directly in StartUML by nesting states. For entry and exit actions, see primitive states.

Semantics. When a compound transition processing reaches a composite state as a target of a transition, it immediately continues to the one initial pseudostate owned by this composite state.

For the semantics of enabled outgoing transitions of a composite state, see the semantics of transitions and primitive states.

Whenever processing enters a composite state, its optional entry expression is executed.

Whenever processing exits a composite state, its optional exit expression is executed.

5.7. Final States

Supported Features. None.

A final state cannot have outgoing transitions, entry or exit expressions.

Modeling in StarUML. A final state is modeled directly as a `FinalState` in StarUML. Its unqualified name can be specified as for any other state.

Semantics. See also the semantics of primitive states and transitions.

When processing of a compound transition reaches a final state, it changes the state of the host object via `HostClass.setState()` to that final state (note that a final state has its name as any other state).

Then, only if there are any enabled triggerless outgoing transitions of the first enclosing composite state, if any (and only of that state, not of the indirectly enclosing states), such an enabled transition is fired and the processing continues over that transition to its target node.

Otherwise, the final state is a steady state and the processing completes.

Implementation. See the implementation of primitive states and transitions.

5.8. Submachines

Supported Features. When an SM needs to include a submachine, it uses the submachine state just as a composite state. That submachine state refers to the definition of the submachine.

A submachine is defined as a separate modeling element (an ordinary SM) outside and independently from SMs that refer to it. The submachine definition SM is also independent from the host class and may reside in a different package.

SOLoist SMs support entry and exit points of submachines, as described in a separate section.

Transitions can have a submachine state as their source and target nodes.

Modeling in StarUML. A definition of a submachine is made as a definition of an ordinary SM. It can reside in a completely different package than an SM that refers to it. For the purpose of code generation, when the definition of an SM is not associated with a host class in the model, the host class

has to be specified through the `HostClass` tagged value for the SM in the SOLoist profile.

To include a submachine, use the `SubmachineState` modeling element.

Warning: StarUML has a bug with generating the XMI file for a model that has an SM referring to a definition of a submachine through the `SubMachine` property in the property panel. As a workaround, instead of referring to a definition through this property, the definition should be referenced with the fully qualified name of the definition through the `SubMachine` tagged value from the SubMachine tag set of the submachine state.

Semantics. When referenced from an SM, a submachine as a whole has the semantics of a composite state, when the transitions targeting the submachine or exiting the submachine outside entry/exit points are concerned. For the transitions to/from entry/exit points, see entry and exit points.

Implementation. At this stage of implementation, the code generated for a submachine is completely embedded in the code generated for the SM that refers to it.

5.9. Entry and Exit Points

Supported Features. SOLoist SMs support entry and exit points only in the context of submachines.

Within a definition of an SM (that can then be referenced from another SM as a submachine), entry and exit points can be *defined*. They have the scope of the entire SM.

Within an SM that refers to another SM as its submachine, entry and exit points of the referenced SM are *referenced*.

Modeling in StarUML. Since StarUML does not support entry/exit points directly, the following workaround is taken.

Within a definition of an SM (that can then be referenced from another SM as a submachine), entry and exit points are defined as simple primitive states stereotyped as `<<entry>>` or `<<exit>>`, respectively. (It is also recommended to use a different color for their symbols in state diagrams to improve readability.) They have a scope of the entire SM (nesting of these states is of no relevance). Each such state should have its name specified, which has to be unique in the scope of the entire SM. These states should not have entry and exit expressions. Entry points must not have incoming transitions, while exit points must not have outgoing transitions.

When an SM with entry/exit points is referenced as a submachine, it is modeled as a submachine state. As its entry/exit points (as a kind of the submachine's interface) are not visible in StarUML, the following workaround is taken:

- outgoing transitions from the submachine state with the stereotype `<<exit>>` are assumed to go out from exit points of the submachine; the referenced exit point's name must be specified in the name property of the transition itself (the points are matched by their unqualified names);
- incoming transitions to the submachine state with the stereotype `<<entry>>` are assumed to go into entry points of the submachine; the referenced entry point's name must be specified in the name property of the transition itself (the points are matched by their unqualified names).

Semantics. Entry and exit points have simple semantics of transient nodes without any additional effects: whenever a transition processing reaches an entry/exit point, it continues to its fired outgoing transition in the referenced submachine or the enclosing SM, respectively, as if the definition of the submachine were directly incorporated into the definition of the enclosing SM.

5.10. Timeouts

The only partial and indirect support for time events, i.e., for objects residing in states with timeouts, is the following. (This lack of support is due to the lack of the footprint of SMs in the UML reflection. For that reason, the SOLoist runtime cannot recognize timed states and host objects that are waiting for timeout in a steady state in a generic way.)

The timeout can be specified for a primitive state in the `timeout` tagged value available in the SOLoist profile. It has to be an expression that results in a long value expressed in miliseconds. This time is the relative time from the moment of entering the state, when the timeout event will be issued.

When a compound transition processing completes and the SM reaches a steady state, it calls back the operation `HostClass.setTimeout()` of the host object. The argument of this call is the timestamp of the timeout (the absolute date and time of expiration), and the fully qualified name of the state. By this, the host object has the opportunity to store the deadline (e.g., in its attribute) when the SM has to be triggered by a timed event (timeout).

When a transition processing exits a state with a timeout, it calls back the operation `HostClass.resetTimeout()` of the host object, with the name of the state as the argument. The host object should then stop the timer, if necessary.

The rest of the mechanism has to be ensured externally by the application and is currently out of scope of SMs. For example, the application can typically allocate a periodic process or thread that can search for all host objects residing in a specific steady state with the expired timeout (i.e., with the *deadline*<=*now*). It can then fire the transition of the SM as with any other trigger for such an object (typically, such a trigger is named `timeout`). In the `HostClass.resetTimeout()` operation, the deadline should be simply cleared then.

6. *HostClass* Interface

To support the SM class with all required features, the host class has to implement the following operations:

```
void   lock ();
void   unlock ();

String getState ();
void   setState (String stateFQName);

void   logStateEntry (String stateFQName);
void   logStateExit  (String stateFQName);

void   setTimeout   (String stateFQName, java.util.Date deadline);
void   resetTimeout (String stateFQName);
```

Description of operations:

- `lock`: called at the beginning of the run-to-completion step; gives the host object an opportunity to lock itself for concurrent access;
- `unlock`: called at the end of the run-to-completion step; gives the host object an opportunity to unlock itself for concurrent access;
- `getState`: called at the beginning of the run-to-completion step to read the current steady state of the host object; should return the fully qualified name (with `::` as a separator) of the state;
- `setState`: called at the end of the run-to-completion step to set the current steady state of the host object; receives the fully qualified name (with `::` as a separator) of the state;
- `logStateEntry`: called at the entrance to each state (composite or primitive); gives the host object an opportunity to trace the transition;
- `logStateExit`: called at the exit from each state (composite or primitive); gives the host object an opportunity to trace the transition;
- `setTimeout`: called at the end of the run-to-completion step, after setting the new steady state; receives the fully qualified name (with `::` as a separator) of the state and the deadline computed by adding the timeout specified for the state to this moment; gives the host object the opportunity to store the deadline (e.g., in its attribute) when the SM has to be triggered by a timed event (timeout).
- `resetTimeout`: called when a steady state with a timeout is exited; receives the fully qualified name (with `::` as a separator) of the state; gives the host object the opportunity to nullify the deadline (e.g., in its attribute) when the SM has to be triggered by a timed event (timeout), or otherwise stop the timer associated with this deadline.