Information and Software Technology 55 (2013) 1726–1740

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

An object-oriented implementation of concurrent and hierarchical state machines

Volker Spinke

Parkstraße 8, 65439 Flörsheim am Main, Germany

ARTICLE INFO

Article history:
Received 1 October 2012
Received in revised form 12 March 2013
Accepted 16 March 2013
Available online 27 March 2013

Keywords:
State machines
UML statecharts
State pattern
Double-dispatch
Code generation
Design pattern

ABSTRACT

Context: State machine diagrams are a powerful means to describe the behavior of reactive systems. Unfortunately, the implementation of state machines is difficult, because state machine concepts, like states, events and transitions, are not directly supported in commonly used programming languages. Most of the implementation approaches known so far have one or more serious drawbacks: they are difficult to understand and maintain, lack in performance, depend on the properties of a specific programming language or do not implement the more advanced state machine features like hierarchy, concurrency or history.

Objective: This paper proposes and examines an approach to implement state machines, where both states and events are objects. Because the reaction of the state machine depends on two objects (state and event), a method known as double-dispatch is used to invoke the transition between the states. The aim of this work is to explore this approach in detail.

Method: To prove the usefulness of the proposed approach, an example was implemented with the proposed approach as well as with other commonly known approaches. The implementation strategies are then compared with each other with respect to run-time, code size, maintainability and portability.

Results: The presented approach executes fast but needs slightly more memory than other approaches. It supports hierarchy, concurrency and history, is human authorable, easy to understand and easy to modify. Because of its pure object-oriented nature depending only on inheritance and late binding, it is extensible and can be implemented with a wide variety of programming languages.

Conclusion: The results show that the presented approach is a useful way to implement state machines, even on small micro-controllers.

1. Introduction

Finite state machines are a clear and concise way to describe the behavior of a system reacting to external events. They are a well-known and widely used technique to describe the dynamics of control systems, protocols or graphical user interfaces. A state machine comprises all permitted states of a system and the allowed transitions between them. Transitions are triggered by events and can be guarded by a condition.

To make the description even more expressive, hierarchies of states have been introduced, which leads to hierarchical state machines. In some cases, actions are to be performed in parallel, which leads to concurrent hierarchical state machines. The paper of Harel [1] gives an introduction to the concepts.

The UML [2] has become the most widespread modeling language used today. It provides a state machine diagram which is a graphical representation of a state machine. The UML state machine diagram combines Mealy and Moore machines. Actions depend on both the active state of the system as well as the triggering event and are associated with the transition from one state to the subsequent state, as in Mealy machines. Additionally, it is possible to define entry and exit actions, as in Moore machines. UML state machine diagrams also allow the hierarchical nesting of states. With this features, UML is capable of modeling a large range of state machines, from simple to very complex.

State machines are an important, not to say essential, way to describe the behavior of reactive systems. They are widely used to implement the control logic of all kinds of software – on a small micro-controller as well as in a large server application.

Unlike classes and objects, current mainstream programming languages, like C++, Delphi or C#, do not support state machines directly. What we are looking for, is a way to implement state machines, which is universally applicable, independent of a special programming language, shows sufficient performance and enables us to make use of the more advanced features like nesting, concurrency and history as well as the advantages of object-orientation.

E-mail address: vs@spinke.de

0950-5849/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2013.03.005

In addition to this, we want an approach to support the UML semantics.

The search for this proposal forms the outline of the paper: at first we look at the traditional approaches and find out, that they have some limitations that make them suboptimal for the intended purpose. The next step is to clarify the requirements of a suitable approach in more detail. As a result of these considerations, we conclude, that applying the well-known object-oriented pattern named double-dispatch to implement state machines should be the solution we are looking for. Then we have a look at the literature to verify this.

Unfortunately, there were no papers found that describe how to implement state machines with the double-dispatch approach. Furthermore, there were no papers found that suggest other approaches that fulfill the requirements set before. Because of this unexpected result, this paper was written to fill this gap.

In the later sections, we describe how to apply the double-dispatch approach to the implementation of state machines and compare it with the traditional approaches. In order to have one representative for each of the traditional approaches, visualSTATE [3] (nested switch/if statements approach, state-event-table approach) and the modified approach of Niaz and Tanaka [4] (based on the state pattern by Gamma et al. [5]) are included in the comparison. In addition to these, the Boost Statechart Library [6] was chosen as a representative of language specific approaches. The Boost Statechart Library is based on lists.

The paper ends with a conclusion and an outlook on the future work.

2. Traditional approaches

In the past, a confusing large number of implementation proposals have evolved. This makes it difficult for a user to choose an appropriate one. At a closer look, there are many similarities, because they basically originate from one of three traditional approaches.

2.1. Nested switch/if statements

The most simple and straightforward approach is to nest two switch statements using scalar variables to represent the states and events. The outer switch statement e.g. selects between different states and the inner one selects between the possible events in this state. The same result can be achieved with if-statements too. Often both are combined: the outer selection is a switch-statement and the inner selection is done by if-statements.

This works well for small and simple state machines but gets cumbersome and confusing very fast, as the number of states and events grow. Besides this, the run-time heavily depends on the way how the compiler translates the switch statement into machine code. Switch statements can be implemented as a series of if statements or by using a jump table. If a series of if statements is used, the runtime is not constant but depends on the active state and the event to be processed and degrades with the number of cases. If a jump table is used, the implementation is similar to the state-event-table approach, described in the next section.

2.2. State-event-table

A more sophisticated approach is to store the transition information in a table. One dimension of the table represents all possible states, the other one all possible events. Using contiguous numbers for both states and events as an index to the table, it is easy to look up the action to perform.

With pointers to functions as table data, the execution time of this approach is fast and does not depend on the size of the table. The run-time for loading the pointer from the table is constant. Nevertheless, this advantage is spoiled to some extend, because in most applications, external events must be mapped to contiguous numbers to access the table. This introduces a search algorithm again, which increases the complexity.

The table approach too gets cumbersome and confusing as the number of states and events grow. The matrix gets large, but is usually sparse. Initialization of the table is complicated and prone to errors if done manually. Nesting is possible, but tedious to implement.

Automatic code generation can overcome these limitations, but usually makes the resulting code practically impossible to read for a human programmer. This aspect is especially important during debugging.

2.3. State pattern

The state pattern published by Gamma et al. [5] is a well-known object-oriented approach for coding state machines. The basic idea is to implement each state as a separate class and each event as a method of this state class. The invocation of a concrete method is done by delegation and late binding.

The state machine is represented by an object which offers methods for all supported events. The methods themselves delegate a call to a local state object. Now, it is easy to change the behavior of the state machine reacting to an event by simply exchanging the state object. Each state object is free to implement the event methods in a different way.

The state pattern has some nice advantages: the execution time is constant (action execution is not taken into account), due to the late binding. State-specific behavior is localized in a single class. This eases debugging and maintenance.

But there is also the other side of the medal: events must be mapped into a method call, which often requires a switch statement or search algorithm again. Further more, the state pattern requires some discipline from the developer and needs a lot of code to write. Changes to the state machine can affect quite a few classes. As van Gurp and Bosch [7] outline, the disadvantages of the state pattern mainly result from the fact, that the only concept explicitly represented is the state. All other elements of a state machine (events, transitions, etc.) are modeled merely implicitly.

Unfortunately, the original state pattern as described in [5] lacks some of the more advanced features of UML state machine diagrams: it is not hierarchical nor does is tell us how to implement entry and exit actions, concurrent states or states with history. Yacoub and Ammar [8,9] present a set of patterns, that extend the state pattern with these features. Also Niaz and Tanaka [4] present an extension to the state pattern. Adamczyk [10] provides an anthology of 23 state machine design patterns many of which are extensions of the state pattern too. Domínguez et al. [11] compiled a table summarizing the features of many approaches, including those mentioned above.

3. Requirements on an alternative approach

The previous section explained the drawbacks of the traditional approaches. But how should an alternative look like?

What we are looking for, is an approach that is first of all fully object-oriented. All externally visible components shall be objects. That means, not only the state machine itself shall be an object, but also its interface to the outside world shall use objects. This leads to the postulation that also the events must be objects. It is not self-explanatory why we should still use enumerated numbers to

represent events. The need to call different methods of a state machine to distinguish external events is also not practical. In real-world systems, an external event is captured e.g. in an interrupt-routine or an event handler of a window system, saved in some kind or other and then forwarded to other parts of the system where it is finally processed. In a program written in an object-oriented language it is only natural to store the occurrence of an event and any accompanying information in objects. All traditional approaches introduce an impedance mismatch to an object-oriented outside world: there is a transformation necessary, a mapping to methods, table entries or scalar values.

An alternative approach shall leverage the advantages of object-orientation also to state machines.

In an object-oriented approach events are objects. This allows events not only to signal that something has happened, but also to carry additional information. It allows events to be grouped to super-classes. This enables state machines to react to a number of events in the same way without having to define separate transitions for all derived events. The other way round, events can be specialized by sub-classes allowing state machines to react to specific events in a special way. This is the equivalent to polymorphic objects.

All state machines shall have a uniform interface that consists of only one method. This method must be able to accept all events that are possible in a system and not only those, which the state machine reacts to.

A human reader must be able to program a state machine that is understandable by others without the need of a code generator. The functionality should be recoverable from the code.

Last but not least, it must allow for easy transformation of a UML state machine diagram into code. Therefore, at least the following features must be supported: hierarchical (composite, nested) states, orthogonal (concurrent) states, entry- as well as exit-actions, guards, shallow and deep history. These are the main components that form the expressive richness of state machines. Other elements of the UML state machine diagram like do activities, choice points and other pseudostates can be emulated with these basic features.

The alternative approach shall be able to process the event objects directly without transforming them to other artifacts. If the states are modeled by objects too, the reaction of the state machine depends on two objects: the current state of the state machine and the triggering event. The mechanism that maps two objects to a reaction is called double-dispatch. A double-dispatch should therefore be a suitable implementation strategy for state machines.

4. Related work

4.1. Tools

There are many commercial and non-commercial tools available, which are able to generate code from state machine specifications.

A brief look at some of the marketing brochures reveals, that in general, these tools make use of one of the traditional approaches mentioned at the beginning (nested switch/if statements, table approach or state pattern) in some way or other, but introduce proprietary variations and extensions.

A closer look at visualSTATE [3] shows, how this works: The user enters a state chart graphically. The tool generates a set of rules, which are stored in several tables. A runtime environment processes these tables and calls one of the action handlers, which are normal C-functions. VisualSTATE is also able to generate code based on nested switch/if statements. The vendor of visualSTATE, IAR Systems, calls this "readable code".

The serious drawback of many tools is, that they do not support some of the more advanced features we are interested in for this paper, like concurrency, hierarchy and history. Even if they can, they do not implement UML semantics or use language features that are non-portable to other programming languages. Sameks C++-approach in [12] is an example of these. It can handle the advanced features but performs transition actions before exit actions. This simplifies the implementation, but is a violation of the UML semantics. It also heavily depends on specific language features and is therefore hard if not impossible to transfer to other programming languages. Also the Walkabout class presented by Palsberg and Jay in [13] falls in this category. Walkabout is implemented using the reflection capabilities of Java, which are not supported in other programing languages, like C++ e.g. That is why such approaches were not investigated further.

4.2. Domínguez et al.

The scientific literature is full of proposals of how to implement state machines in various programming languages and how to generate code from state machine specifications. An in-depth overview and comparison of 53 proposals is provided by Domínguez et al. [11]. Instead of citing many of the papers examined by Domínguez et al. here again, the interested reader is highly advised to have a look at this extensive work.

An interesting finding of Domínguez et al. is, that the state transition process in all of the proposals they examined is based either on a switch-statement or a state-table. Other possibilities (like lists or double-dispatch) are not mentioned at all.

4.3. Niaz and Tanaka

Niaz and Tanaka [4] extended the state pattern in Gamma et al. [5] with history, composite and concurrent states. So, they overcome the limitations of the original pattern mentioned earlier. Unfortunately, their approach is not UML compliant too. The UML specification requires an event to be processed by the innermost state which defines a transition that is triggered by this event. If an event has triggered a transition, it must not trigger any other transitions in super states any more (except for default transitions, of course). The event has been 'consumed'.

In contrast to this, the approach of Niaz and Tanaka processes every transition that is triggered by a specific event, regardless, whether the event has already triggered a transition in a nested state or not. This is due to the fact, that Niaz and Tanaka did not implement any mechanism to decide whether an event has to be processed or not, if it already has been consumed in an inner state.

The approach of Niaz and Tanaka results in fast code execution, but is limited to the implementation of special cases of state machines only. Behavioral inheritance as described in the books by Samek [12,14] is not possible with their original approach. The possibility to define transitions triggered by any supported event in any state, regardless of whether the state is nested into an other one or not, is essential.

Because this is easy to add, the author of this paper extended the approach of Niaz and Tanaka by this feature. The state pattern is based on delegation. The events are mapped to method calls of the outermost state object. The outermost state object itself delegates these method calls to the active state object. To make nested states possible, the approach of Niaz and Tanaka continues this delegation to the innermost state. In order to enable a state to decide if it is allowed to process the event, simply a boolean return value is added to the methods executing the events, which tells the caller, whether the event has been processed or not. Now, an outer state is able to decide, if it is allowed to process the event

or not. With this little extension, the approach of Niaz and Tanaka is able to implement UML semantics too.

4.4. Chauvel et al.

The double-dispatch approach models states as well as events explicitly as objects. This is a mayor difference to other patterns, that model only states as objects, like the state pattern. If both states and events are objects, double-dispatch can be applied to select the right behavior according to the event and the current state. Chauvel and Jézéquel [15] just mention this briefly in their paper but neither give any details of how to implement it nor do they compare this approach to others.

4.5. The Boost Statechart Library

The rational of The Boost Statechart Library [6] mentions double-dispatch as a considered but rejected implementation. It simply states, that double-dispatch "scales badly". It is neither self-explanatory nor explained in the rational why an algorithm that merely performs two consecutive virtual function calls should not scale. The given handcrafted example as well as the cited Acyclic Visitor [16] pattern makes heavy use of templates, RTTI and casts. It is probably the casts, that do not scale.

In addition to this, the GoF Visitor pattern [5] is criticized because it is necessary to recompile the whole state machine, if an event is added. This is true, but usually no problem. In most applications, the number of events is limited and known beforehand, because they reflect external incidents the system shall react to. It is usually the reaction to these external triggers but not the triggers themselves that change during development. If it is really necessary that many developers work on a large state machine implementation simultaneously, it is easy to break the state machine classes as well as the event classes of the approach presented in this paper into several compilation units that can be worked on independently. Certainly, it is still necessary to merge newly added events into the header file of the state class later when all parts are put together, but this is easily done with the support of current version management tools and integrated development environments. The heavy criticism on the necessity to recompile – especially in [16] – appears exaggerated in most cases. Today, we have development tools at hand, that support refactoring.

5. The double-dispatch approach

5.1. Key problem

The key idea of a state machine is that its reaction depends on two parameters: the active state and a triggering event. Guarded by a Boolean expression, both parameters determine a transition to a subsequent state. While traversing the transition, an action is performed.

The process of mapping a message to a specific method at runtime is known as dynamic dispatch. In many object-oriented languages, like C++, Delphi and C#, the concrete code that is invoked by a virtual function call, depends on the type of the object at runtime. As only *one* object determines which code is going to be executed, they are known as *single* dispatch calls.

Because the transition to be executed and its corresponding action depends on *two* parameters (state and event), single-dispatch languages have no language features to implement state machines directly.

5.2. Concept

The solution to this shortcoming is known as double-dispatch. Generally explained, a *double*-dispatch is a mechanism that dispatches a virtual function call to different concrete functions depending on the runtime types of *two* objects involved in the call – in case of a state machine, these are the active state and the triggering event.

Because there is no direct support for double-dispatch in single-dispatch languages, we need to emulate the mechanism by code. This is easily done in the following way, similar to ping-pong: (Code snippets are in C++)

Step (1) At first, we must bring the ball into play: a method of the state object is called with an event object as parameter:

```
void State::dispatchEvent(Event *event)
{
    event->processFrom(this);
}
```

Step (2) The ball is then pushed to the opposite party, the event class. Now, it is the event's turn to push back the ball: Provided, the type of the parameter event is of type `EvModeBtn`, the previous call from `State::dispatchEvent()` results in the invocation of the method:

```
void EvModeBtn::processFrom(State *state)
{
    state->processEvent(this);
}
```

Step (3) Provided, the type of the parameter state is of type `StateCooler`, the previous call from `EvModeBtn::process-From()` results in the invocation of the method:

```
void StateCooler::processEvent(EvModeBtn *event)
{
    //perform a transition here
}
```

As a consequence, the resulting function call `StateCooler::processEvent(EvModeBtn *)` depends on both the type of the state object and the type of the event object. If the event changes to an object of type `EvSpeedBtn`, a function named `StateCooler::processEvent(EvSpeedBtn *)` is called. If the state object changes from an instance of class `StateCooler` to an instance of class `StateHeater` and the event object is of type `EvSpeedBtn`, the resulting function call is `StateHeater::processEvent(EvSpeedBtn *)`.

In order to use late binding (virtual function calls) on the state object, the state class hierarchy needs a common base class. Same applies to the event class hierarchy too. If default handlers in the

base class of the states are provided, is is not necessary to implement all possible pairs of state and event types, but only those, that are really needed in the application. This reduces the amount of code to write considerably.

5.3. Advantages

The double-dispatch is easy to implement and depends only on two object-oriented mechanisms: inheritance and late binding. These are available in all OO programming languages.

States and events are both objects and not only numbers. This enables the user to adapt the classes to the specific needs of the application, simply by sub-classing, without affecting the mechanism.

Only those transitions, which are defined in the state machine diagram need to be coded. All other combinations are handled by the default handlers in the state base class.

There is no need to define states and events as contiguous numbers, as it is with the table approach. The state and event classes can be reused in different contexts. If a context object is provided, states and events do not need to contain any data and therefore even the objects themselves can be reused if implemented with the flyweight pattern (see Flyweight Pattern in [5]). The cost for creation and deletion of the state and event objects is payed only once at program initialization and is zero during run-time. This is especially advantages in embedded systems with limited resources, where memory management is often restricted.

There is no mapping required between events and object methods (and therefore no switch statement or search algorithm). Because of the default handlers in the base classes, the state machine can handle all supported events, without any precautions. In contrast to this, in a table approach, special care must be taken, to limit the indexes for the table look-up to existing values. This is especially important for C and C++ which do not have automatic range checking by default.

Execution time is constant and independent of the number of states and events. To find the concrete transition function, two consecutive virtual function calls are performed, each of which has a constant run-time. (Action execution is not taken into account.)

States, events and transitions have a one-to-one mapping from the state machine diagram to their implementation. This is important in maintenance and debugging.

Because the transitions are made explicit as methods, it is even feasible to reconstruct the state machine from the code.

Last, but not least, the code is readable and understandable for a human reader.

5.4. Disadvantages

There must be a common base class for all states which defines default handlers for each event in the system. Otherwise polymorphism is not possible and the code does not compile. As a result, the base class `State` depends on all defined events. Adding a new event, requires to add a handler in the `State` class as well, but the necessary changes are small. Because `State` is a base class, all dependent code has to be recompiled. As the number of events is usually limited and known beforehand, this dependency is uncritical.

In the other way around, the event class hierarchy is completely independent of the state class hierarchy. Adding a new state does not cause any changes to the event class hierarchy.

5.5. Main difference to other approaches

In contrast to most other solutions, the approach presented in this paper, reifys *both* states and events as objects but not the transitions. Why is this rational?

Events are entities that are part of the problem domain. A reactive system detects events and processes them e.g. by means of a state-machine which determines the appropriate reaction to the actual event. In an object-oriented design, it is only natural to model the events as well as the state-machine as objects. This leads to the requirement, that the state-machine must provide an interface that can handle these event-objects.

States are only internal entities of the state-machine, so it seems not necessary to model them as objects. As a matter of fact, the classical approaches like the switch/if-statements approach as well as the state-table approach do not model the states as objects as the state-pattern does. Nevertheless, a state typically models a situation during which some (usually implicit) invariant condition holds true. Because of this, it makes sense to model the states as separate self-contained entities – in other words: objects – too.

The transitions are neither visible from outside the state machine nor are they relevant from an outside point of view. Only the behavior of the state machine, which is the reaction to an event, is essential. Therefore, it does not matter how these internal elements are implemented. There are no apparent advantages to model the transitions as objects. A reason that could justify to implement the transitions as objects is the possibility to change the behavior of the state machine at run-time similar to the Strategy Pattern in [5]. By changing the behavior, a different state machine is being build which leads to the question why not to model and build different state machines for each case and exchange the state machine object rather than the transitions.

These considerations lead to the design as presented in this paper. The use of the double-dispatch mechanism is a direct consequence of the reification of both states and events because it allows an easy mapping from the current state and the triggering event to an action without artificial transformations.

6. Implementation

6.1. A brief example

To proof the usefulness of the proposed approach, a brief example was implemented. The system models an air conditioner. It is controlled by three buttons: Power, Mode and Speed. Pressing the Power button starts the system to operate. There are two modes of operation: heating and cooling. With the Mode button, the user switches between these modes. The fan of the air conditioner can be set to two speeds: low or high. With the Speed button the user selects the speed of the fan. The Mode and the Speed buttons shall be disabled unless the air conditioner is operating. While in operation, pressing the Power button stops the system. Pressing the Power button again restarts the system in the same mode and speed it was in, before being stopped.

The state machine diagram in Fig. 1 represents the control logic of this air conditioner system. If pressed, the three buttons generate the events `EvPowerBtn`, `EvModeBtn` and `EvSpeedBtn`. The controller reacts by calling one of the action functions `EnableButtons`, `CoolerOn`, `FanLow`, etc.

The class `AirConCtrl` presents the whole state machine to the application program. `AirConCtrl` encapsulates two states named `Stopped` and `Operating`. The state `Stopped` is a simple state that is also the initial state.

The state `Operating` is an orthogonal state with two regions. It contains two sub state machines, named `Mode` and `Speed`, which control the mode of the system and the speed of the fan respec-

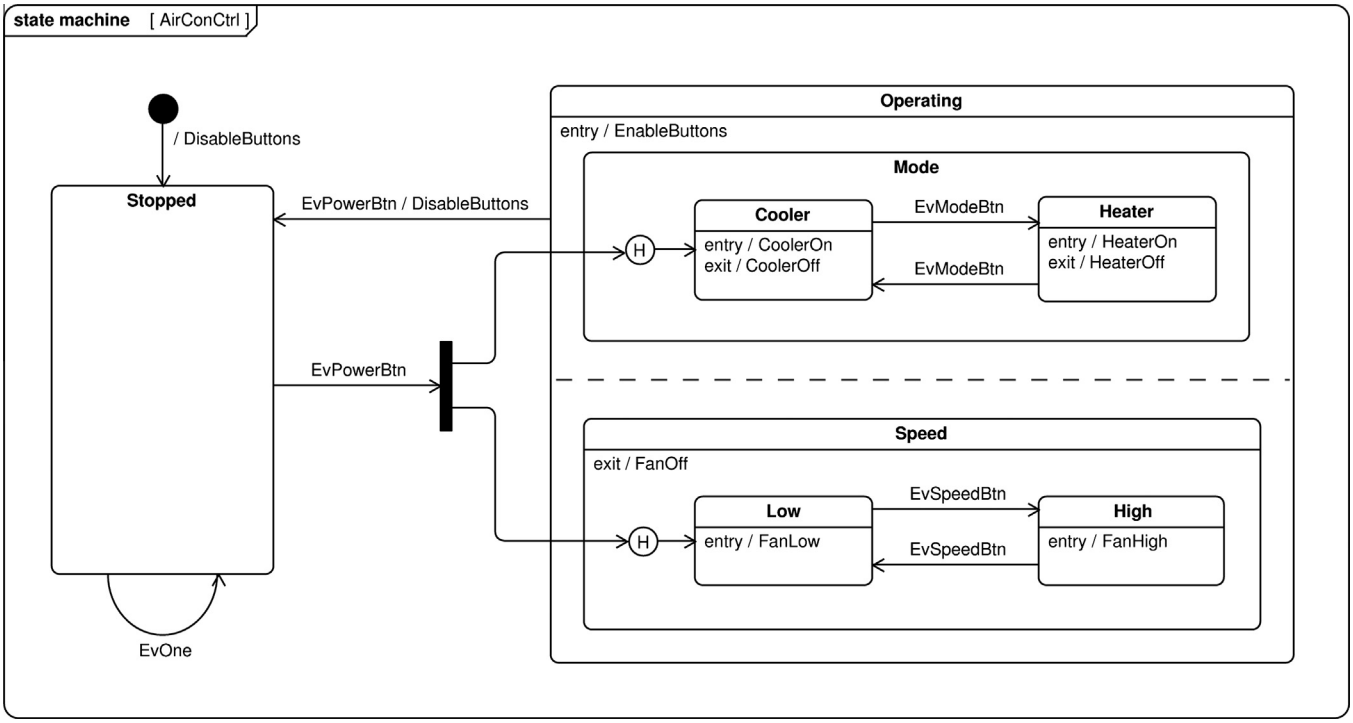

Fig. 1. State machine for an air conditioner.

tively. They operate in parallel. The event `EvModeBtn` switches between the modes `Heater` and `Cooler`, whereas the event `EvSpeedBtn` switches between the fan speeds `Low` or `High`.

The event `EvOne` was introduced for testing purposes only.

The example is similar to the one presented by Niaz and Tanaka in [4]. It was chosen because it is uncomplicated and easy to understand but includes some of the more advanced features like concurrent states and history. Unlike Niaz and Tanaka, the author of this paper considers it best, to model the pre- and postconditions of the states with entry and exit actions. The `Operating` state e.g. requires the mode and speed buttons to be enabled. As a consequence, the `Operating` state ensures that the buttons are enabled, by processing an entry action `EnableButtons`, which enables the buttons. If the `Cooler` state is left, the cooler shall be off in any case. So, it is the responsibility of the `Cooler` state to define and process an exit action, which turns off the cooler. The same principle applies to all other states. That is why the example makes heavy use of entry and exit actions. Unfortunately, we end up with an example that has no actions associated with transitions any more. To show how they are implemented too, it was decided to break the rule with respect to the `Stopped` state. Because of educational purposes only, disabling the buttons is done as an action associated with all transitions to the `Stopped` state. No doubt, disabling the buttons in an entry handler of the `Stopped` state, is a much better solution from an engineering perspective.

6.2. Basic structure

Fig. 2 shows the static structure of the implementation of the air conditioner example from Fig. 1.

The basic element of the implementation is a state which is represented by a class named `State`. A `State`-object does not contain any further elements, but can have entry- and exit-actions assigned to it, as well as internal transitions. `State` has a method named `dispatchEvent()` that processes an incoming event.

In contrast to `State`, an object of class `StateMachine` is a state that contains further elements: states or state machines and transitions between them. Only one of the inner states of a state machine can be active at a time. The attribute `activeSubState` holds a reference/pointer to this currently active `State`-object. Because `StateMachine` is derived from `State`, the active sub-state can be a simple state or a complex state machine. The hierarchy of states in the state machine diagram is mapped into a tree of `State` and `StateMachine` objects by means of the Composite Pattern in [5]. The `dispatchEvent()`-method of class `StateMachine` propagates the incoming event to its active sub-state. By this, the event is dispatched recursively down the tree of states, until it reaches a simple state, that does not have any further inner elements. Because a `StateMachine` is a `State`, it can have entry- and exit-actions assigned to it as well as internal transitions.

The concurrent nature of a state is implemented by the class `ParallelState`. It holds a number of regions, which are references/pointers to `State`-objects, that are active at the same time. The sole purpose of `ParallelState` is to dispatch incoming events to all of its regions. As `ParallelState` is also derived from `State`, it can be part of the composite tree of states and have entry- and exit-actions as well as internal transitions.

The application specific state classes simply derive from one of the general classes `State`, `StateMachine` or `ParallelState` and add the application specific behavior.

The events of the system are modeled by a class named `Event`. This serves as a superclass. The actual events, that are applied to a `State` by the `dispatchEvent()` method, are all derived from this superclass. They form a simple class hierarchy with no further dependencies.

An important aspect of the design is that every level of the hierarchy of states is independent of the other ones. Each state only needs to know its ancestor. The ancestor of a state is the state that contains this state. In other words: A state is a sub-state of its ancestor. In addition to this, each `StateMachine`-object certainly needs to know all of its sub-states. In case of the example shown in

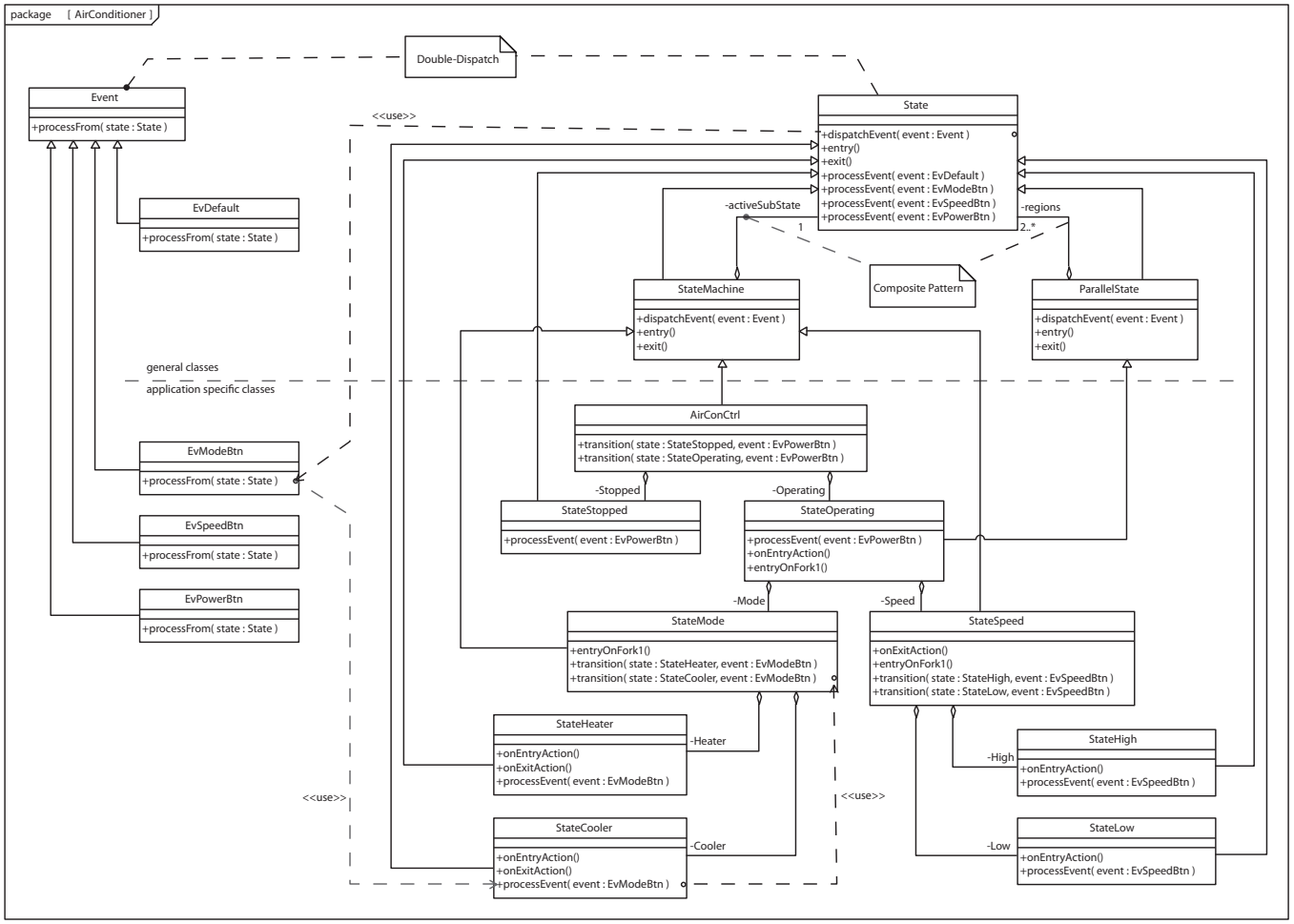

Fig. 2. Static structure of the air conditioner example.

Fig. 1, the outermost `StateMachine`-object that represents the whole state machine is named `AirConCtrl`. It aggregates state `Stopped` and state `Operating` only. An event is simply passed on to the next lower level of the hierarchy of states. A `StateMachine`-object calling the `dispatchEvent()`-function of its `activeSubState` does not know and does not need to know what type the `activeSubState` is and if it has an inner life or not. So, `AirConCtrl` simply passes the event to `Operating` e.g. That `Operating` has a complicated inner structure is not the business of `AirConCtrl`. It is the responsibility of `Operating` to ensure that the event is processed accordingly.

This structure has some important implications. First, each state can easily be extended. If there are only transitions within this extended state (as it is in the example with regard to `Operating`), the ancestor is not affected at all. Second, transitions are methods of a state and must always be performed by the least common ancestor (LCA) of the source and target states involved in that transition. The LCA of two states is that state which contains both states as sub-states. The double-dispatch results in a call of a method of the currently active state. It is now the responsibility of this active state either to perform an internal transition itself or call a method of his ancestor to perform a transition from that state if one is defined. Third, because a `StateMachine`-object does not know if a certain sub-state is a simple `State` or another `StateMachine`, it must delegate the entry of a state to an appropriate entry-handler. The newly entered sub-state must handle the entry according to his structure, e.g. enter its sub-state by an appropriate entry-handler. In case of the example shown in Fig. 1, state `Operating` must provide an entry-handler to realize the fork, because `AirConCtrl` does not know about `Mode` and `Speed` and therefore cannot restore the previously active states.

The principle of delegating up and down the hierarchy of states is fundamental to the approach presented in this paper. It enables the user to hide the internals of the involved objects but also to extend the structure. A consequence of this is the possibility to put parts of the state machine in different compilation units and modify them independently.

6.3. The code

The following subsections show the relevant parts of the code.

6.3.1. Events

At first, we need to define the event class hierarchy. The class `Event` is the base class. We then derive a basic event from it, that we need to implement the state machine: `EvDefault`. The application specific events are derived in the same way in a separate file (see Listing 1).

The implementation is straight forward and always the same for all events (see Listing 2).

6.3.2. States and state machines

Next, we need to define the state class hierarchy. State machines and orthogonal states are states too. So, they are derived from the base class `State` (see Listing 3 State.h).

The implementation is shown in Listing 4 State.cpp.

State.h and State.cpp give us some kind of ‘library’, which is simply used in the application-specific code later.

```
class Event
{
public:
  Event(void) {return;}
  virtual ~Event(void) {return;}

  virtual bool  processFrom(State *);
};

class EvDefault : public Event
{
public:
  virtual bool  processFrom(State *state);
};

extern EvDefault       aEvDefault;
```

Listing 1. Events.h.

```
bool EvDefault::processFrom(State *state)
{
  return state->processEvent(this);
}
```

Listing 2. Events.cpp.

The heart of the code is the `dispatchEvent()` virtual function of class `StateMachine`. It delegates an incoming event down the hierarchy of nested states to the active sub-state of the state-machine. The `dispatchEvent()`-function always returns a boolean value, which indicates, whether the parameter event has been processed or not. Remember, that in C/C++ (as in other languages like Delphi too) a boolean expression is evaluated from left to right. This means, that the right part of the or-operator (||) is only evaluated, if the left part has been evaluated to false. If the left part is true instead, the result is always true and therefore the right part does not need to be computed any more and in fact is not computed at all. We use this feature, to consecutively dispatch the event to the next level, if it has not been processed.

The most nested `State` object calls the function `processFrom` which looks for transitions from that state, either internal or outgoing. If there is none defined, the default-handlers of class `State` return false, indicating, that the event has not yet been processed. Now, the call hierarchy returns to the `dispatchEvent()` virtual function of class `StateMachine` which itself calls `processFrom` looking for an internal or outgoing transition from that state.

At last, the next line looks for a default-transition to be performed within the state machine. Because the event objects are defined as flyweights (they have no attributes at all), they can safely be reused. Instead of constantly creating and destroying new event objects all the time, predefined event objects are used here, one for each event class. This is especially advantages in micro-controller-applications where dynamic memory management is restricted. This can easily be changed, if an application requires an event to carry additional information.

Derived states do not need to override the `dispatchEvent()` virtual function (neither that of `State` nor that of `StateMachine`), but simply provide `processEvent`-handlers for the particular events they want to process.

There are three different ways to enter a composite state. They are shown in Fig. 3.

A transition ending on the border of a composite state means that the sub-state that is denoted as the initial state becomes the active sub-state. This is implemented by the `StateMachine::entry()` virtual function. At first it calls an `onEntryAction()`-handler if one is defined. The second step is to initialize the `activeSubState`-attribute with the initial sub-state of the state. At last, the entry-action of the newly set active sub-state is invoked. This completes the entry-sequence.

```
class State
{
public:
  State(void);
  virtual ~State(void);

  virtual bool  dispatchEvent(Event *event);
  virtual void  entry(void);
  virtual void  exit(void);
  virtual void  restoreDeepHistory(void);
  virtual void  restoreShallowHistory(void);

  //general events
  virtual bool  processEvent(EvDefault *) {return false;}

  //application specific events
  virtual bool  processEvent(EvPowerBtn *) {return false;}
  virtual bool  processEvent(EvModeBtn *) {return false;}
  virtual bool  processEvent(EvSpeedBtn *) {return false;}
  virtual bool  processEvent(EvOne *) {return false;}

protected:
  virtual void  onEntryAction(void);
  virtual void  onExitAction(void);
  virtual void  setInitDefaultState(void);
  virtual void  setShallowHistoryDefaultState(void);
  virtual void  setDeepHistoryDefaultState(void);

private:
  State(const State &);
  State& operator=(const State &);
};

class StateMachine : public State
{
public:
  StateMachine(void);
  virtual ~StateMachine(void);

  virtual bool  dispatchEvent(Event *event);
  virtual void  entry(void);
  virtual void  exit(void);
  virtual void  restoreDeepHistory(void);
  virtual void  restoreShallowHistory(void);

protected:
  State *activeSubState;
  State *previousSubState;

private:
  StateMachine(const StateMachine &);
  StateMachine& operator=(const StateMachine &);
};

class ParallelState : public State
{
public:
  ParallelState(int noOfRegions);
  virtual ~ParallelState(void);

  virtual bool  dispatchEvent(Event *event);
  virtual void  entry(void);
  virtual void  exit(void);
  virtual void  restoreDeepHistory(void);
  virtual void  restoreShallowHistory(void);

protected:
  int    noOfRegions;
  State **regions;

private:
  ParallelState(const ParallelState &);
  ParallelState& operator=(const ParallelState &);
};
```

Listing 3. State.h.

A transition to a history state needs a special treatment. A separate entry-handler-function must be defined that handles this case. The entry-handler first calls the `onEntryAction()`-handler as always, then restores the `activeSubState` to its previous value and finally enters the restored sub-state. The following snippet shows the relevant code (see Listing 5):

```
bool State::dispatchEvent(Event *event)
{
  return event->processFrom(this);
}

void State::entry(void)
{
  this->onEntryAction();
}

void State::exit(void)
{
  this->onExitAction();
}

void State::onEntryAction(void)
{
  return;
}

void State::onExitAction(void)
{
  return;
}

void State::setInitDefaultState(void)
{
  return;
}

void State::setShallowHistoryDefaultState(void)
{
  return;
}

void State::setDeepHistoryDefaultState(void)
{
  return;
}

void State::restoreDeepHistory(void)
{
  return;
}

void State::restoreShallowHistory(void)
{
  return;
}

StateMachine::StateMachine(void)
: activeSubState(NULL)
, previousSubState(NULL)
{
  return;
}

StateMachine::~StateMachine(void)
{
  return;
}

bool StateMachine::dispatchEvent(Event *event)
{
  bool retval = activeSubState->dispatchEvent(event) ||
       event->processFrom(this);
  aEvDefault.processFrom(this);

  return retval;
}

void StateMachine::entry(void)
{
  this->onEntryAction();
  this->setInitDefaultState();
  this->activeSubState->entry();
}

void StateMachine::exit(void)
{
  this->activeSubState->exit();
  this->previousSubState = this->activeSubState;
  this->activeSubState = NULL;
  this->onExitAction();
}

void StateMachine::restoreDeepHistory(void)
{
  this->activeSubState = this->previousSubState;
  if (NULL == this->activeSubState)
  {
    this->setDeepHistoryDefaultState();
  }
  this->activeSubState->restoreDeepHistory();
}

void StateMachine::restoreShallowHistory(void)
{
  this->activeSubState = this->previousSubState;
  if (NULL == this->activeSubState)
  {
    this->setShallowHistoryDefaultState();
  }
}

ParallelState::ParallelState(int noOfRegions)
: noOfRegions(noOfRegions)
, regions(NULL)
{
  regions = new State*[noOfRegions];

  for (int i=0; i<noOfRegions; i++)
  {
    regions[i] = NULL;
  }
}

ParallelState::~ParallelState(void)
{
  delete[] regions;
}

bool ParallelState::dispatchEvent(Event *event)
{
  bool retval = false;

  for (int i=0; i < noOfRegions; i++)
  {
    bool result = (regions[i])->dispatchEvent(event);
    retval = result || retval;
  }

  retval = retval || event->processFrom(this);

  aEvDefault.processFrom(this);

  return retval;
}

void ParallelState::entry(void)
{
  this->onEntryAction();

  for (int i=0; i < noOfRegions; i++)
  {
    (regions[i])->entry();
  }
}

void ParallelState::exit(void)
{
  for (int i=0; i < noOfRegions; i++)
  {
    (regions[i])->exit();
  }

  this->onExitAction();
}

void ParallelState::restoreDeepHistory(void)
{
  for (int i=0; i < noOfRegions; i++)
  {
    (regions[i])->restoreDeepHistory();
  }
}

void ParallelState::restoreShallowHistory(void)
{
  for (int i=0; i < noOfRegions; i++)
  {
    (regions[i])->restoreShallowHistory();
  }
}
```

Listing 4. State.cpp.

In order to do this, the exit-handler `StateMachine::exit()` stores the `activeSubState` that is becoming inactive, because the state is left, in an attribute. This enables the `restoreShallowHistory()` and the `restoreDeepHistory()` func-

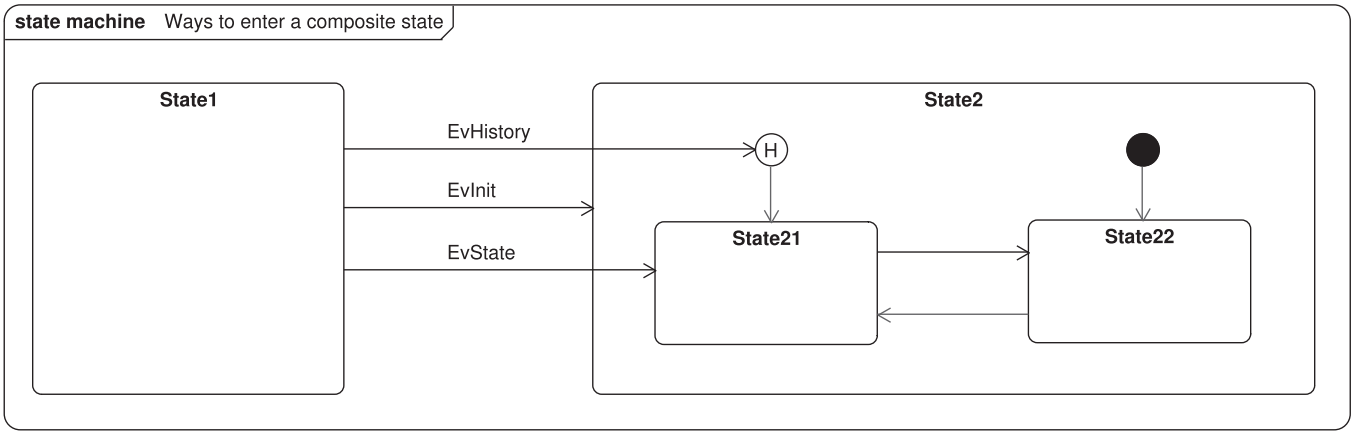

Fig. 3. Ways to enter a composite state.

```
void StateMode::entryOnFork1(void)
{
    this->onEntryAction();
    this->restoreShallowHistory();
    this->activeSubState->entry();
}
```

Listing 5. Operating.cpp.

tions to restore the previously active sub-state. With this strategy, the implementation of history simply turns into an initialization procedure of the active sub-state, which it naturally is.

A transition ending on a particular state is handled in the same way. A special entry-function simply sets the sub-state becoming active after the transition is executed like the previous code shown restores the history state.

The method that implements the transition must choose the appropriate entry-function according to the destination of the transition.

As already described, there must be default handlers defined for each event in the system, otherwise the polymorphism does not work. This makes class `State` depend on all events, as is easily seen in the class definition. Introducing a new event is simply done by defining a class for the new event and adding one handler-function in the `State` class. Afterwards, all of the code needs to be recompiled. The advantage is, that all state machines now support the new event. It can be passed to any state machine. No special care must be taken, to pass only those events to a state machine that this particular state machine processes. If the state machine does not define a reaction to this new event, nothing happens.

6.3.3. Application code

With this small 'library' presented above, it is easy to write application code like in Listing 6:

How is a transition performed? Assume, that the state machine in Fig. 1 is in state `Cooler`. An event `EvModeBtn` is processed by a call to `AirConCtrl::dispatchEvent()`. `AirConCtrl` inherits method `dispatchEvent()` from `StateMachine` which dispatches the event to `Operating` which in turn dispatches it to the `Cooler` state object. Because `Cooler` is an atomic state, the default `dispatchEvent()`-virtual function of class `State` performs the double-dispatch, as described in Section 5.2. The flow of control is also visualized in Fig. 2 by the dotted lines marked with stereotype «use» The result is a call to `StateCooler::processEvent(EvModeBtn *)`. This function initiates the transition by calling a transition function of its ancestor state (the state one level above in the state hierarchy) named `StateMode::transition(StateCooler *, EvModeBtn *)`. As shown in Listing 6, this transition function calls the exit-handler of its active sub-state (which is still state `Cooler`), then performs an action, if one is defined (there is none in the example), changes its active sub-state to a new sate (which is state `Heater` in the example) and then performs an entry action on the new active sub-state. Because the event has now been fully processed, true is returned, which prevents further processing of the event, as described earlier.

The code in Listing 6 also gives examples of how to handle entry, exit and initialization events. The given code should be self explanatory. The rest of the state machine shown in Fig. 1 is implemented in the same way.

Listing 7 gives an example of how to program a transition with an action associated that is controlled by a guard. The pattern is the same as before. Instead of the comment in the previous listing, the action `EnableButtons()` is called. The transition is executed only if the function `TempIsNormal()` that acts as a guard returns true. After the transition was executed, the transition handler returns true to indicate, that the event `EvPowerBtn` has been processed. Otherwise it must return false to enable other states in the hierarchy to process the event.

7. Comparison

7.1. Runtime

7.1.1. Test programs

To compare the run-times of the different approaches, a couple of test programs were written as described in Table 1. Each test program implements the state machine described in Fig. 1.

The nested switch/if statements approach, as well as the table approach are represented by visualSTATE generated code. The state pattern approach is represented by the approach presented by Niaz and Tanaka [4]. Their Java code was converted to C++ and modified as described in Section 4.3. Language specific approaches are represented by an implementation with The Boost C++ Statechart Library. The double-dispatch approach as presented in this paper comes in three flavors. The first one is the version shown by the code examples in this paper. The second one combines the double-dispatch with delegation similar to the state pattern. The third version is optimized like the approach presented by Niaz and Tanaka. It implements only the necessary functionality directly in the appropriate handlers, as a code generator would do it.

7.1.2. Test cases

Each test program performs three tests as described in Table 2. Test A generates randomly 1000 events and processes them 1000 times. All four events defined in the example in Fig. 1 have equal

```
void StateCooler::onEntryAction(void)
{
  this->context->CoolerOn();
}

void StateCooler::onExitAction(void)
{
  this->context->CoolerOff();
}

bool StateCooler::processEvent(EvModeBtn *event)
{
  return this->ancestor->transition(this, event);
}

StateMode::StateMode(AirConCtrl *ancestor, AirConCtrlCtxt
    *context)
: ancestor(ancestor)
, context(context)
, Heater(new StateHeater(this, context))
, Cooler(new StateCooler(this, context))
{
  return;
}

StateMode::~StateMode()
{
  delete this->Cooler;
  delete this->Heater;
}

void StateMode::setInitDefaultState(void)
{
  this->activeSubState = this->Heater;
}

void StateMode::setShallowHistoryDefaultState(void)
{
  this->activeSubState = this->Cooler;
}

void StateMode::entryOnFork1(void)
{
  this->onEntryAction();
  this->restoreShallowHistory();
  this->activeSubState->entry();
}

bool StateMode::transition(StateCooler *, EvModeBtn *)
{
  this->activeSubState->exit();
  //no action defined
  this->activeSubState = this->Heater;
  this->activeSubState->entry();

  return true;
}

bool StateMode::transition(StateHeater *, EvModeBtn *)
{
  this->activeSubState->exit();
  //no action defined
  this->activeSubState = this->Cooler;
  this->activeSubState->entry();

  return true;
}
```

Listing 6. Operating.cpp.

probability. Test A is to test the over-all performance of the different algorithms. Test B repeats events with no transition in state `Stopped`, while Test C does the same in state `Operating`. Both tests are to check the effect of the number of nested states on the performance of the algorithms.

Each test program processes one million events. All test programs have empty action handlers. The action handler functions simply return immediately without doing anything. As a result, only the performance of the control algorithms are measured.

```
bool AirConCtrl::transition(StateStopped *, EvPowerBtn *)
{
  if (this->context->TempIsNormal())
  {
    this->activeSubState->exit();
    this->context->EnableButtons();
    this->activeSubState = this->Stopped;
    this->Stopped->entry();

    return true;
  }

  return false;
}
```

Listing 7. AirConCtrl.cpp.

7.1.3. Results

To carry out the measurements, several compilers and compiler settings were tried on different operating systems. The actual runtimes and program sizes fluctuate considerably depending on the compiler and its option settings. As this is not a compiler comparison, we do not go into details here, but emphasize that the influence of the compiler *can* outweigh the influence of the algorithm. GCC results are reported, because this is the most widely accessible reference platform.

Furthermore, on a preemptive multi-tasking operating system, it is not possible to measure the runtime of a program accurately. That is why it is useless to claim something like "our code is 58.53% more efficient than an other one" as other papers do. The runtimes stated in the tables may fluctuate up to about 10% from pass to pass. The interesting point for this paper is the relative performance of the tested algorithms against each other, which is stable, not their absolute performance.

Table 3 shows the results on Linux compiled with GNU CC 4.4.3 with options -Os (Optimize for size) and -fno-exceptions and -fno-rtti. The measuring was performed on a PC equipped with a 1.8 GHz AMD Athlon processor.

It seems that the double-dispatch is a little bit slow compared to other approaches. This is of no wonder. Remember, that the state hierarchy forms a tree. The code shown in this paper performs a double-dispatch on each level of that tree of states. This is obviously more time consuming than e.g. the state pattern that simply delegates the event down the hierarchy. We come back to this point later in this paper.

It is interesting that the table approach shows a relatively poor performance. This is due to the run-time system necessary to process the tables. The results comply with articles by IAR. The table approach has advantages, if the state machine gets large. In this case, the code necessary to process the tables keeps the same, while the tables grow in size only moderately. This may result in smaller code sizes compared to other approaches. According to IAR, the nested switch/if-statements approach is usually faster than the table approach. The results shown in this paper confirm this.

Each algorithm has its strength and weaknesses. It is usually easy to find cases, in which a given algorithm performs superb and others in which it performs weak. The double-dispatch approach proposed in this paper as well as the State Pattern based approach proposed by Niaz and Tanaka [4] are recursive algorithms, that are susceptible to the number of nested states. Tests B and C clearly indicate this. The performance of the State Pattern approach as well as the double-dispatch approach is excellent compared to the nested switch/if statements approach, if the nesting depth is low (one nesting level in Test B). The run-times increase noticeably, if the nesting depth becomes higher (three levels in Test C), while the run-time of the nested switch/if statements approach as well as the table-approach keeps nearly constant.

Table 1
Test programs.

Name	Description
DDA	double-dispatch approach as presented in this paper
DDD	double-dispatch approach combined with delegation
DDO	double-dispatch approach optimized version
VST	visualSTATE version 6.1 generated code: table approach
VSR	visualSTATE version 6.1 generated code: readable approach (nested switch/if statements)
TAN	State-Pattern approach as presented by Niaz and Tanaka converted from Java to C++
BST	implementation with The Boost C++ Libraries version 1.50

Table 2
Test cases.

Name	Description
Test A	Generate randomly 1000 events and process them 1000 times (all events have the same probability)
Test B	Events with no transition in state Stopped (repeat sequence EvModeBtn, EvSpeedBtn)
Test C	Event with no transition in state Operating (send event EvOne)
	Each test processes exactly one million events

Table 3
Runtimes on Linux DDA.

	Test A		Test B		Test C	
DDA	173 ms	100%	60 ms	100%	171 ms	100%
VST	256 ms	148%	136 ms	227%	125 ms	73%
VSR	122 ms	71%	106 ms	177%	101 ms	59%
TAN	84 ms	49%	25 ms	42%	46 ms	27%
BST	778 ms	450%	163 ms	272%	334 ms	195%

Table 4
Runtimes on Linux DDD.

	Test A		Test B		Test C	
DDD	150 ms	100%	48 ms	100%	134 ms	100%
VST	256 ms	171%	136 ms	283%	125 ms	93%
VSR	122 ms	81%	106 ms	221%	92 ms	69%
TAN	84 ms	56%	25 ms	52%	46 ms	34%
BST	778 ms	519%	163 ms	340%	332 ms	248%

Table 5
Runtimes on Linux DDO.

	Test A		Test B		Test C	
DDO	74 ms	100%	27 ms	100%	50 ms	100%
VST	256 ms	346%	137 ms	507%	125 ms	250%
VSR	122 ms	165%	105 ms	393%	92 ms	184%
TAN	84 ms	114%	25 ms	93%	46 ms	92%
BST	773 ms	1045%	163 ms	604%	328 ms	656%

The poor results of the Boost Statechart Library certainly speak for themselves.

The previous results raise the question why not to combine the double-dispatch with delegation like in the state pattern? The second double-dispatch implementation DDD implements this. It performs a double-dispatch at the highest level and then delegates the event down the hierarchy of states. To do this, it is necessary to implement additional delegation functions which make the code more complex. Nevertheless, the results in Table 4 show an improvement compared to the previous results.

The comparison is still misleading because the double-dispatch approach is the only one that uses objects as events, not numbers as the other approaches (except Boost) do. The comparison of two numbers is always faster than any thing else. In addition to this, all test programs except DDA, DDD and BST are highly optimized versions. It is practically impossible to change these state machines without the use of the code generator that produced the test programs. So, the question is what happens, if the double-dispatch approach is implemented in the same highly optimized way as the other three test programs VST, VSR and TAN? The result is shown in Table 5.

The performance boost is considerable. The implementation DDO is comparable with those of VST, VSR and TAN. The 'library' shown in the listings above was moved into the different handlers and stripped to the minimal functionality that is necessary to implement the state machine diagram in Fig. 1. As a result, the convenience and ease of use that the small library presented above introduced is gone. Even for small modifications of the state machine, a lot of code needs to be changed, as it is for the other approaches too.

Table 6 shows the results on an ARM7 target, compiled with IAR Embedded Workbench 5.20 with optimization "high size" in Processor mode "ARM". On the ARM7-target, best results are achieved, when optimization "high size" in ARM mode is used. This option results in fast and compact code. The measuring was performed on a NXP LPC2468 micro-controller running at 48 MHz.

The results are similar to those on Linux and show, that the proposed approach is applicable on small micro-controllers too.

7.2. Code size

Table 7 shows the code size of each test program in bytes.

The test programs VSR and VST are mainly pure C code. It seems that the GNU C-Compiler is capable of optimizing this much better than other compilers can do. Tests on Windows with Visual C++ showed a code size that is roughly twice as large as that of the GNU compiler. The proportions for the C++-based test programs are similar to each other between the compilers.

As already seen before, the Boost approach falls out of the ordinary also with regard to the code size. It compiles to up to five times larger code than the proposed approach. Size matters, especially on micro-controllers, where memory is usually very limited. This is also true for applications running on more powerful embedded microprocessors with an underlying operating system like Windows CE or Embedded Linux.

The error message reported by the IAR cross compiler ("Embedded C++ does not support run-time type information.") clearly reminds us, that a template library is not intended for use on small micro-controllers.

As a result, we can document, that the code size for the approach presented in this paper is slightly bigger than that for traditional approaches, but small enough that it is still useful.

7.3. Maintainability

The nested switch/if-statements approach does not scale well – not only with regard to runtime, but especially related to maintenance. In larger state machines, it gets difficult to find the right place to do modifications.

Generating the source code seems to be an elegant way out of this difficulty, but then all modifications must be made using the tool. This procedure is inconvenient and sometimes impractical. Without the tool, it gets difficult to change the source code. Even

Table 6
Runtimes on ARM7.

	Test A		Test B		Test C	
DDA	10988 ms	100%	3802 ms	100%	13896 ms	100%
DDD	10091 ms	92%	3521 ms	93%	11021 ms	79%
DDO	4674 ms	43%	1667 ms	44%	4104 ms	30%
VSR	10238 ms	93%	8291 ms	218%	8083 ms	58%
VST	24780 ms	226%	13458 ms	354%	11937 ms	86%
TAN	5177 ms	47%	1989 ms	52%	4083 ms	29%

Table 7
Size of each test program in Bytes.

	PCLinux[a]		ARM7[b]	
DDA	30 098	100%	24560	100%
DDD	30 308	101%	25404	103%
DDO	24 698	82%	23052	94%
VST	15 667	52%	20537	84%
VSR	15 437	51%	20541	84%
TAN	23 815	79%	21676	88%
BST[c]	151 617	504%	–[d]	–

[a] Compiled with GNU CC 4.4.3 with options -Os (Optimize for size) and -fno-exceptions and -fno-rtti.
[b] Compiled with IAR Embedded Workbench 5.20 with optimization "high size" in Processor mode "ARM".
[c] Requires RTTI and Exceptions turned on to compile.
[d] IAR Embedded Workbench 5.20 reported: Error[Pe878]: Embedded C++ does not support run-time type information.

worse, manual changes are lost, if the code is regenerated later. Round-trip engineering is usually not provided by the tools.

Furthermore, the nested switch/if-statements approach requires the source code of the state machine to be completely in one file. This is a disadvantage, because it is easy to lose track of things in large files.

The state-event-table approach is more easy to handle with regard to manual changes. With proper names of the handler-functions (e.g. a combination of the state and the event, the function is handling), it is clear where to make modifications. Nevertheless, if things grow, it is also very easy to be off the track in large tables.

The state pattern is quite verbose – it needs a lot of code to write and a lot of classes to handle. The introduction of a new element (a new state, a new event or a new transition) may result in many small changes at many different places. This can be handled with some discipline, but is not really maintenance friendly. If the state machine grows, this makes the state pattern cumbersome too.

The double-dispatch approach presented in this paper, makes it easy to split up the source code of a large state-machine into several files. The approach itself, does not impose any restrictions on the organization of the source code, as long as the compiler is able to translate a compilation unit. If wanted, each state and event class could even be placed in a separate file.

It is easy to develop parts of a hierarchical state machine independently and reuse them in a different context. Because the states are objects, it is irrelevant for an outer state machine using them, what happens inside. Each state provides a `dispatchEvent()`-method. What happens behind this interface, is the sole responsibility of the state object. If it is a state machine, it delegates the event to its active sub state. If it is a parallel state, it delegates the event to its regions. If it is an atomic state, it initiates the double-dispatch mechanism to look for a transition to perform. In the air-conditioner example, the top-level state machine does not know and does not need to know at all that the state `Operating` has an inner life.

Because the events are implemented as classes, it is easy to group them by defining super classes. Transitions on a group of events, need only be coded once, simply by defining a transition processing an event of the super class. As as a result, it is not necessary to define separate transitions for every single subclass. To illustrate this, consider a calculator as an example. There are 10 events defined, one for each digit. A transition shall be performed on pressing any of the digits. Instead of defining ten separate transitions for every single digit event, simply define a superclass for all digits and then implement a transition processing this superclass event. Because any subclass is also of the type of its superclass, the transition is performed irrespective of which digit is pressed.

Another advantage of using classes to represent events is, that events can hold additional information (like a time stamp e.g. or a temperature in the air conditioner example). The user is free to use all possibilities of object-orientation. To add additional information to an event is not possible in other approaches (nested switch-statements, state-event-table), because events are represented there by arbitrary insignificant numbers.

Stepping through the code in a debug session, reveals that it is easy to navigate through code that uses the double-dispatch approach. If meaningful names for states and events are used, the programmer always knows where he is and how the program got there.

This is due to the regular structure of the approach, which also supports maintenance. Although adding a new element like a new transition may require changes at more than one place, the necessary changes are very local. They usually affect only the state and its surrounding state machine, but not the whole class hierarchy.

7.4. Portability – use of special language features

It was one of the main aims of this work, to keep the approach independent of special language features. Special language features, like templates and runtime type information in C++ or reflection in Java, are usually not portable to other programing languages in a consistent way. A good example for this, is the approach of Samek presented in his first book [12] which is based on C++. It is a mixture of objects, inheritance and switch-statements. The implementation relies very much on specific language features of C++ like function pointers and type-casts. The author of this paper tried to port Sameks approach to Delphi but gave up because of constant nagging about type errors by the compiler.

Special language features also tend to be extremely expensive. Palsberg and Jay [13] generalized the visitor pattern to a "Walkabout class", using the reflection capabilities of Java. The performance penalty is of the order of 300, compared to manual implementations.

Similar results are seen for C++-templates. Already the compilation process needs much longer than approaches omitting templates. The run times of the test programs implemented with the Boost Statechart Library were always the longest and up to 36 times higher than those of other approaches. Embedded C++, a subset of C++ intended for programming small micro-controllers, does not support templates at all. Furthermore, the template syntax is hard to read and debug.

A special disadvantage of the Boost Statechart Library is that it is not possible to group events by simply defining super classes, as the documentation explains. In a general approach, there is no reason, why this should not be possible, because factoring out a super class, is a common and natural process in object-oriented software development. In addition to this, the sequence of exit actions is not UML compliant. The Boost Statechart Library processes the exit handlers top-down the state hierarchy instead of in bottom-up order.

A useful state machine pattern is independent of a specific programming language and can therefore be implemented with many different target languages. This is the case for all of the traditional approaches mentioned at the beginning of this paper. The approach proposed here follows this deliberate tradition.

7.5. How to implement features not described yet

In Listing 7 we showed how to implement guards. This section briefly describes how to implement other more advances features of the UML that are missing yet.

A Join is easy to realize similar to a guard. Instead of an external condition, the if- statement checks if some states are active before the transition is performed. Final states are easy to realize as their behavior is similar to simple states. Do-activities of states can be emulated by entry and exit actions. Junction-points and dynamic choice points can be implemented with arbitrary complex if-statements within the transition functions that decide on the actions to perform and the target state. Deferred events can be implemented with an external message queue. In many cases, it is also possible to avoid deferred events by using orthogonal states. The implementation of a message queue as well as the use of threads is out of the scope of this paper.

7.6. Element-based overview of state machine elements

Domínguez et al. provide an element-based comparison with all of the approaches examined by them. They write: "The goal of this comparison is to analyze which state machine elements are considered by each proposal and the strategy followed in order to implement these elements as code structures." This section shows how the approach proposed in this paper fits in the pattern system by Domínguez et al. (see Table 10 in [11]). By this, it indicates the commonalities and differences to the other approaches.

The approach presented in this paper, implements the state machine elements in the following way:

Context Class. Class.
Current State. Attribute holding a reference to a state object.
Simple State. The proposed approach uses a specific class for each state in the state machine specification. It derives these state classes from a base class, but the bases-class is not and cannot be purely abstract. The state machine is composed by aggregation, not by inheritance.
State transition process. Double-dispatch.
(It is noteworthy that the only two mechanisms found by Domínguez et al. in 28 proposals described in 53 papers are the switch-statement and the state-table structure.)
Events. A base class for all events plus a concrete sub-class for each event in the system.
Guards. If-statement in the transition handler method, that decides, if the transition is to be executed or not.
Actions. An action class with a method for each action (A3).
Composite State Simple. A state class that aggregates sub-states. Events are delegated to the active sub-state.
Composite State Orthogonal. A state class that aggregates sub-states. Events are delegated to all sub-states.
Fork. Normal transition plus special entry action that activates and enters more than one sub-state.
Join. If-statement in the transition handler method.
(Implemented like a guard that checks if the state machine is in the states where the join originates from.)
Choice. If-statements in the transition handler method that determine the target state. Alternative: State with outgoing default transitions controlled by guards.
History. Normal transition plus special entry action that has to restore the history states from an attribute. Shallow and deep history are supported.
Activities. A specific method in the corresponding state class (A3). Do-activities can be emulated by entry- and exit-actions.

A default initialization transition may have an action associated with it. This is not possible with the Boost Statechart Library. A composite state can use default initialization, deep and shallow history simultaneously in accordance with the UML semantics.

In addition to the table presented by Domínguez et al. (Table 10 in [11]), the element-based overview demonstrates, that the approach proposed in this paper shares some minor commonalities with other approaches but has some major differences in its structure and the way it operates, so that it can be considered as substantially different to the approaches presented in the comparison by Domínguez et al.

8. Conclusion and future work

An object-oriented approach for state machine implementation has been presented. States, as well as events are modeled by classes. Transitions are methods of the state classes. Because the actual transition to perform depends on the type of two objects (active state and triggering event), the double-dispatch mechanism is used for the mapping.

The results proof that this approach is a useful way to implement state machines. It is easy to code, easy to understand and therefore easy to maintain and debug. Furthermore, it executes fast, compared to other popular approaches. Because of its pure object-oriented nature, the presented approach can be easily reused and extended.

The fact, that this approach relies only on inheritance and late binding, makes it suitable for use with any object-oriented language. To verify this, examples were coded in C++ and Delphi. They all worked as expected.

As it is not only a variation or combination of existing approaches, but has unique characteristics in its structure and the way it operates, the author concludes that applying the double-dispatch technique to implementing state machines, can be considered as a suitable approach, that should gain more attention as a fourth solution in addition to the three traditional approaches.

It is notable that authors proposing an approach for state machine implementation usually omit information on how their algorithm scales. As a future work, the author of this paper plans to further examine how the algorithms scale with regard to run-time and code size if the number of states, transitions and events get large.

Acknowledgment

Thanks to Dr. Iftikhar Azim Niaz for providing the Java source code of the example in his paper [4].

References

[1] D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (1987) 231–274. <http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf>.
[2] Unified Modeling Language Specification (UML), Object Management Group, 2012. <http://www.omg.org> (last visited 14.08.12).
[3] visualSTATE – A Set of State Machine Design, Test and Implementation Tools, IAR Systems AB, Version 6.1. <http://www.iar.com>.
[4] I.A. Niaz, J. Tanaka, An object-oriented approach to generate java code from UML statecharts, International Journal of Computer and Information Science (IJCIS) 6 (2) (2005) 83–98. <http://www.iplab.cs.tsukuba.ac.jp/paper/journal/niaz_IJCIS2005.pdf>.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995. ISBN 0-201-63361-2.
[6] Boost C++ Libraries, Version 1.50. <http://www.boost.org>.
[7] J. van Gurp, J. Bosch, On the implementation of finite state machines, in: Proceedings of the IASTED International Conference, 3rd Annual IASTED International Conference Software Engineering and Applications, Scottsdale, Arizona, USA, 1999.
[8] S.M. Yacoub, H.H. Ammar, Finite state machine patterns, in: Proceedings of Third European Conference on Pattern Languages of Programming and Computing, EuroPLoP'98, Bad Irsee, Germany, 1998.
[9] S.M. Yacoub, H.H. Ammar, A pattern language of statecharts, in: Proceedings of Fifth Annual Conference on the Pattern Languages of Programs, PLoP'98, Allerton Park, Illinois, USA, 1998b.
[10] Adamczyk, The anthology of the finite state machine design patterns, in: Proceedings of the Pattern Languages of Programs Conference (PLoP), 2003.
[11] E. Domínguez, B. Pérez, A.L. Rubio, M.A. Zapata, A systematic review of code generation proposals from state machine specifications, Information and Software Technology 54 (10) (2012) 1045–1066. http://dx.doi.org/10.1016/j.infsof.2012.04.008. ISSN 0950-5849.
[12] M. Samek, Practical Statecharts in C/C++. Quantum Programming for Embedded Systems, CMP Books, 2002. ISBN 1-57820-110-1.
[13] J. Palsberg, C.B. Jay, The essence of the visitor pattern, in: Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference, 1998. COMPSAC '98., 1998, http://dx.doi.org/10.1109/CMPSAC.1998.716629.
[14] M. Samek, Practical Statecharts in C/C++, Second Edition. Event-Driven Programming for Embedded Systems, Newnes – An Imprint of Elsevier, 2009. ISBN 978-0-7506-8706-5.
[15] F. Chauvel, J.-M. Jézéquel, Code generation from UML models with semantic variation points, in: Proceedings of MODELS/UML'2005, Springer, Montego, 2005.
[16] R.C. Martin, Acyclic Visitor, Object Mentor, 1996. <http://www.objectmentor.com/resources/articles/acv.pdf>.