Towards Integration of State Machines and Object-Oriented Languages

Ole Lehrmann Madsen
The Danish National Centre for IT Research
Computer Science Department, Aarhus University
Åbogade 34, DK-8200 Århus N, Denmark
Ole.L.Madsen@{cit.dk,daimi.au.dk}
Tel.: +45 8942 5670, Fax: +45 8942 2443

Abstract

The goal of this paper is to obtain a one-to-one correspondence between state machines as e.g. used in UML and object-oriented programming languages. A proposal is made for a language mechanism that makes it possible for an object to change its virtual bindings at run-time. A state of an object may then be represented as a set of virtual bindings.

One advantage of object-orientation is that it provides an integrating perspective on many phases of software development, including analysis, design and implementation. For the static set of OO language constructs there is almost a one-to-one correspondence between analysis/design notations and OO programming languages. No such correspondence exists for the dynamic aspects, but the proposed state-mechanism is a contribution to a better correspondence. The proposal is based on previous work by Antero Taivalsaari and compared to the more complex features for changing object behavior as found in CLOS, Smalltalk, and Predicate Classes, it is simple and efficient to implement.

Keywords: Object-oriented analysis and design, object-oriented languages, state machines.

1. Introduction

State machines are often used to describe the dynamic behavior of objects and most main stream object-oriented analysis and deign notations, like UML [BRK99], includes a notation for state machines, often in the form of statecharts [Har87]. A state machine is associated with an object when the behavior of the object may naturally be divided into different logical states. In theory the behavior of an object of course always depends on its state since the values of its state variables may be said to define its state, but here we by state mean a logical state at a higher level. An object has a set of interface functions (virtual methods) and to support state machines, the state must be encoded into these functions. This can be done by using a simple state variable or by using the more fancy state pattern [GHJV95].

States may also be supported by changing the class type of an object and thereby the implementation of the virtual functions defined for the object. Changing dynamic class membership is a complicated operation and various proposals ranging from a complete general mechanism in Smalltalk [GR89] and CLOS [Kee89] to e.g. Predicate classes [Cha95] exists. In this paper we present an extension of the BETA language based on the very simple idea of Taivalsaari [Tai93] of being able to specify a number of alternative virtual bindings for a class. A simple change of the dispatch pointer may then be used to change the virtual bindings of an object. In addition we present an extension of the

mechanism that makes it possible to combine state machines and inheritance. Inheritance of state may be used to represent nested states and inheritance of state machines may be used to classify state machines. The latter is based on the notion of virtual class patterns from BETA.

In the next section we discuss the background and motivation for a further integration of state machines within an object-oriented language. We then present an example of a state machine and goes through the various techniques for implementing state machines in an OO language. The main emphasis here is on the use of the state pattern, since the proposed extension of BETA may be considered as a direct support for state patterns. We then describe the proposed extension of BETA based on [Tai93] followed by a description of the implementation. Finally it is shown how the proposed state classes can be combined with inheritance. The proposal is presented as an extension of BETA, but may be incorporated into most OO languages. Knowledge of BETA [MMN93,Mad94,Mad98] will be an advantage, but a short summary of the used BETA syntax will be given.

2. Background and motivation

One of the advantages of object-orientation is that it provides an integrating perspective on many phases and activities of software development. Object-orientation provides a conceptual framework and a common set of abstract language constructs that can be applied in analysis, design and implementation. This is in contrast to traditional methodologies like Structured Analysis and Structured Design (SA/SD) where different concepts and languages were used in analysis, design and implementation. Object-orientation has also proved to be useful for data modeling in databases and for designing and implementing distributed systems.

In mainstream OO methodologies the Unified Modeling Language (UML) is often recommended for analysis and design and popular programming languages like C++ [Str86] and Java [GA98] are used for implementation. One important difference between design notations and programming languages is that the former have a graphical syntax and the latter have a textual syntax. For the core set of OO languages constructs: class, subclass, virtual function, etc., design notations and programming languages can express the same structures and the difference between the two sets of notations for this core set are of a syntactic form. I.e. graphical in contrast to textual syntax.

Most modern software methodologies [Boo91, Zü98] recommend approaches that are iterative, experimental and/or evolutionary. This is in contrast to traditional methodologies based on the waterfall method. With the modern approaches to software development, the software developers will alternate between analysis, design and implementation more or less frequently depending on the nature of the application domain, the application being developed and the actual methodology being applied.

In order to support the transitions between analysis, design and implementation, several vendors offer CASE tools that support code generation from analysis/design diagrams to programs. It is well known that major changes to the design is often applied in the implementation phases which implies that tool support for the reverse process is also needed. That is a CASE tools should also support so-called reverse engineering where design diagrams are updated/generated from the (modified) programs. For many years, most CASE tools had little support for reverse engineering, but this has slowly been improved. There are, however, in practice still many problems with support for reverse engineering in many CASE tools. The code generated from design diagrams is often filled with annotations to ease the reverse engineering process and this implies that the code is longer and more difficult to read than it needed to be. Lately the techniques for reverse engineering have, however, improved.

The above discussion referred to the situation in main stream OO. For the Mjølner System [KLMM94] a different approach has been taken based on the BETA language. BETA is an object-oriented language designed to support analysis, design and implementation. For BETA a main design goal has been that the language should be well suited to model phenomena and concepts from the application domain. BETA may be viewed, as a successor to Simula [DNM68], which was originally, developed in order to write simulation programs, but later generalized to a programming language. The emphasis on modeling may be traced back to the requirements for writing simulation programs.

Presentation of class structures in the form of diagrams may be useful in many situations during a software development process[1]. For this reason a graphical syntax was developed for the overall class structures of BETA. The editors of the Mjølner System may then present a textual as well as a graphical view of a set of classes (or program). The editors are incremental in the sense that a change to the textual representation is immediately reflected in the graphical diagram and vice versa. In addition the editors support a so-called round trip reverse engineering where a design diagram may be generated from the program. In this way the Mjølner System provides strong support for alternation between analysis, design and implementation.

The overall principle in the Mjølner System is that BETA is used for analysis, design and implementation. The experience with the Mjølner System is that it simplifies the development process to use one language for design and implementation since less time is used for the transitions between diagrams and programs.

The almost one-to-one relationship between design notations and programming languages does only exist for elements for describing the static class structures. When it comes to describing the dynamic structure of a system there is much less similarity. Programming language uses algorithmic structures like assignments, control structures, procedures, and concurrent process. The design notations uses state machines and interaction diagrams.

Interaction diagrams are useful for understanding the dynamic behavior of a system. They may be used as tools to help develop an understanding of the dynamic behavior and they may be used to describe this dynamic behavior when a design has been created. Interaction diagrams exists in two forms: sequence diagrams and collaboration diagrams. Sequence diagrams shows examples of calling sequences between objects. Interaction diagrams shows the calling (message sending) relations between objects. Collaboration diagrams can often be displayed from the CASE tool if the code is available. Sequence diagrams can be generated for a design if a suitable simulation/execution mechanism is available [AL99].

State machines are used to describe objects that have a behavior which changes over time in the sense that the actions performed for a given method differs depending on the state of the object. There is no clear definition of this since most objects act depending on their current state. There are a number of techniques for mapping state machines into OO languages. These include using tables, statements, and the state pattern, but there is no tight integration between state machines and programming languages as is the case for the static class structures discussed above. This implies that the problems with code generation and reverse engineering discussed above are present.

For the Mjølner System the use of the same language for design and implementation has proved to be successful for the static class structures. The purpose of this paper is to discuss to what extent a similar integration between state machines and OO programming languages can be made. We first briefly summarize the well approaches of using tables, algorithms and the state pattern. Then we suggest a modification of BETA that makes it

---

[1] The purpose of this paper is not to discuss the advantages of graphical versus textual presentation of designs and programs. It is thus assumed that there is a need for both forms of presentation.

possible to specify one or more bindings for the virtual functions of a class and to change bindings dynamically. This makes it possible to have direct support for state machines in an object. The new mechanism may be viewed as a simple and restricted version of the more powerful means for changing class membership of an object as available in Smalltalk and CLOS or the dynamic inheritance feature of Self [US87] or the predicate classes of Craig Chambers. The state-mechanism proposed for BETA is simple and efficient to implement and it makes it possible to have a direct integration of state machines and programs. In addition we think that it is a simple alternative to the more complex features for changing class/object behavior.

3. State machine for watch

In this section we will introduce an example of a state machine (adapted from [EP98]) which will be used in the following. Consider the digital watch shown in Figure 1. The watch has two buttons, a **mode** button and a **inc** button. Initially the watch displays the time of the day. When the **mode** button is pressed, the hours will be displayed. A subsequent push on the **inc** button will increment the hours. A push of the **mode** button will display the minutes and a subsequent push of the **inc** button will increment the minutes. Finally, a push of the **mode** button will return the watch to a state where the time is displayed.

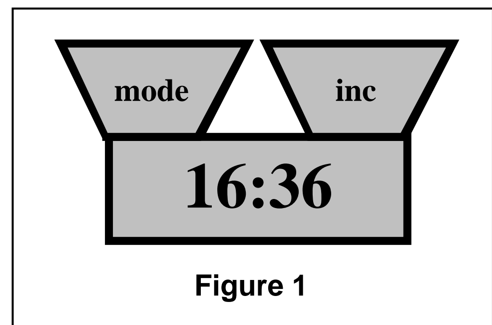

Figure 1

The behavior of the watch is modeled by the state machine shown in Figure 2. The notation for the state-machines is statecharts as used in UML. The round boxes represents states, the arrows represents transitions between states. At each arrow an expression of the form **event**/action describes the event that gives rise to the transition between the states and the action performed when a transition is made.

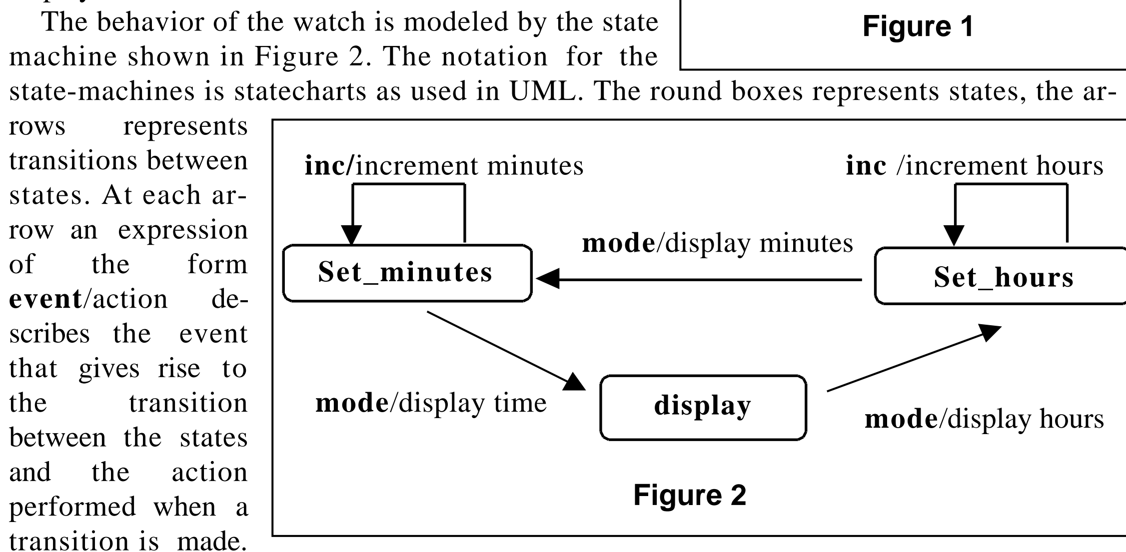

Figure 2

The watch state machines has three states (Display, Set_hours, and Set_minutes), two events (mode and inc), and five actions (display hours, display minutes, display time, increment hours, and increment minutes).

3.1 State machine implementation using tables and algorithmic statements

A well-known technique to implement a state machine is to use a two dimensional table indexed by the current state and the next event. This technique may be a good choice in many situations, but we find that the techniques discussed below are better suited to OO languages and we will not discuss it further.

The perhaps most common technique for implementing a state machine for an object is to use a variable to hold the current state of the object and let each transition correspond to a virtual function. The implementation of a virtual function then uses the

state variable to select the proper actions and determine the next state. A class `Watch` implemented in this way is shown in Figure 3.

```
Watch:
  (# LCD: @ DigitalDisplay;
     state: @ integer; (* Display, Set_hours, Set_minutes *)
     mode:<
       (#
       do (if state
            // Display then LCD.display_hours; Set_Hours -> state
            // Set_hours then LCD.display_minutes; Set_minutes -> state
            // Set_minutes then LCD.display_time; Display -> state
          if)
       #);
     inc:<
       (#
       do (if state
            // Display then skip
            // Set_hours then LCD.incHours
            // Set_minutes then LCD.incMinutes
          if)
       #);
     init:< (# do Display -> state #)
  #)
```

Figure 3

A few comments on the BETA syntax used above:

- The BETA pattern construct is a unification of class, procedure and other abstraction mechanisms. A pattern may be used as a class or procedure. A pattern that is primarily used as a class is called a *class pattern* and a pattern that is used as a procedure is called a *procedure pattern*. Patterns may be *virtual* or *non-virtual* and they may be arbitrarily *nested*. In addition patterns are *values* that may by assigned to variables.
- The construct `Watch: (# ... #)` is an example of a class pattern and corresponds to the notion of class as in most other OO languages.
- The construct `mode:< (# ... #)` is an example of a virtual procedure pattern and corresponds to virtual functions in Simula, C++ and Java. The attributes `inc` and `init` are also virtual procedure patterns.
- The attributes `LCD` and `state` are variables that denote instances of `DigitalDisplay` and `integer` respectively. The `@` indicates that the objects are part objects, which means that they are allocated when the enclosing `Watch` object is allocated.
- The construct `(if ... if)` corresponds to a case-statement or switch-statement.
- Assignment has the form `Display -> state` where the value `Display` is assigned to the variable `state`.
- An expression of the form `LCD.display_hours` is an invocation of `display_hours` in the `LCD` object.

3.2 State machine implementation using state pattern

Encoding the state in a variable and using conditional expressions to select the proper actions may not be the best way to implement a state machine. The state pattern is a technique for using subpatterns and virtual functions to implement state. The idea is to have a class hierarchy corresponding to the states of the state machine and have different implementations of the virtual functions for each state. The state of the object is then represented by an instance of state classes. The methods of the class propagate the calls to the actual state object. This technique is shown in Figure 4.

```
Watch:
  (# LCD: @DigitalDisplay;
     state: ^ WatchState;
     mode:< (# do state.mode #);
     inc:< (# do state.inc #);
     init:< (# do &Display[] -> state[]; this(Watch)[] -> state.init #)
  #);
WatchState:
  (# mode:< (# do inner #);
     inc:< (# do inner #);
     init:< (# W: ^Watch enter W[] do W[] -> theWatch[] #);
     theWatch: ^Watch
  #);
Display: WatchState
  (# mode::< (# do LCD.display_hours; &Set_hours[] -> theWatch.state[] #);
  #);
Set_hours: WatchState
  (# mode::< (#do LCD.display_minutes; &Set_minutes[] -> theWatch.state[]
#);
     inc::< (# do LCD.incHours #)
  #);
Set_minutes: WatchState
  (# mode::< (# do LCD.display_time; &Display[] -> theWatch.state[] #);
     inc::< (# do LCD.incMinutes #)
  #);
```

Figure 4

A few more BETA constructs are used in Figure 4:

- The `^` in `state: ^WatchState` means that `state` is a variable reference (pointer) to instances of `WatchState`.
- The expression `this(Watch)[]` is a reference to `self` as in Smalltalk.
- The construct enter `W[]` describes an input parameter to `init`.
- The `[]` in expression like `W[]` means that the value of `W[]` is a reference to the object referred by `W` as opposed to the value (copy) of the object denoted by `W`. An expression `W` without `[]` means the value (copy) of `W`. Consult [MMN93] for the exact details.
- The `&` in `&Display[]` means `new` as in C++. I.e. a new instance of `Display` is generated.
- A pattern `Display: WatchState(#...#)` describes that `Display` is a subpattern of `WatchState`.
- **Inner** is a mechanism for method combination. In this paper **inner** can be ignored and taken as a substitute for abstract virtual functions. For a real explanation, see [MMN93].
- The construct `::<` means a binding of a virtual pattern from the superpattern.

The advantage of the state pattern is that there is a (sub) class corresponding to each state and that dispatch of actions and state changes are factored out in subclasses. It is possible to vary the state pattern in a number of places:

- An object may be pre-allocated for each state in order to avoid allocating a new one at each state change.
- Instead of a backpointer to the `Watch` object, the reference to the `Watch` object may be passed as an argument to the virtual function calls in `WatchState`. This makes the state objects "state less", which means that they can be used for more than one `Watch` object.
- In languages with nested classes (like Simula, BETA and Java) the state classes may be nested in the `Watch` class. This will eliminate the need for the back pointer and the state classes may be encapsulated as an internal part of the `Watch` class. On the other

hand, it is not possible to reuse the state objects for several `Watch` objects. Since we in the following are proposing a new language mechanism that directly supports nested state patterns, an implementation based on nested state patterns is shown in Figure 5.

```
Watch:
  (# LCD: @DigitalDisplay;
     state: ^ WatchState;
     mode:< (# do state.mode #);
     inc:< (# do state.inc #);
     init:< (# do &Display[] -> state[]; #);
     WatchState:
       (# mode:< (# do inner #);
          inc:< (# do inner #);
       #);
     Display: WatchState
       (# mode::< (# do LCD.Display_hours; &Set_hours[] -> state[] #);
       #);
     Set_hours: WatchState
       (# mode::<  (# do LCD.display_minutes; &Set_minutes[] -> state[] #);
          inc::< (# do LCD.incHours #)
       #);
     Set_minutes: WatchState
       (# mode::< (# do LCD.display_time; &Display[] -> state[] #);
          inc::< (# do LCD.incMinutes #)
       #);
  #)
```

Figure 5

4. Direct modeling of state change in objects

In this section a mechanism for changing the state of an object is introduced. It is based on direct support for the state pattern using nested classes, and the following observations:

- A state class is a collection of virtual function implementations.
- Each state class defines an alternative implementation of virtual functions, and each such implementation represents a state of the object.
- An object may change state by replacing the current virtual function implementations by another implementation of the virtual functions.

In order to support state classes directly, it is proposed that a class pattern may define one or more possible bindings of its virtual functions. A class pattern then has the form shown in Figure 6 where `P` has three virtual functions, `f1`, `f2`, and `f3`. In addition two sets of bindings of the virtual functions of `P` are defined. These are specified by

```
S1: STATE(# ... #)
```

and

```
S2: STATE(# ... #)
```

```
P: (# f1:< ...;
      f2:< ...;
      f3:< ...;
      S1: STATE
          (# f1::< ...;
             f2::< ...;
             f3::< ...;
          #);
      S2: STATE
          (# f1::< ...;
             f2::< ...;
             f3::< ...;
          #);
      ...
   #)
```

Figure 6

When an instance of `P` is generated, the bindings of the virtual functions are as specified by the declarations of `f1`, `f2`, and `f3` at the outermost level of `P`. This is the same as if

there were no S1 and S2. The virtual bindings of a P-object may be changed by executing the imperative:

```
S1## -> this(P)._STATE
```

or

```
S2## -> this(P)._STATE
```

In Figure 7, then class pattern Watch is implemented using the proposed state-mechanism.

```
Watch:
  (# LCD: @DigitalDisplay;
     mode:< (# do INNER #);
     inc:< (# do INNER #);
     init:< (# do LCD.display_time; Display## -> this(Watch)._STATE  #)
     Display: STATE
      (# mode::< (#do  LCD.display_hours;  Set_hours## -> this(Watch)._STATE
#);
      #);
     Set_hours: STATE
      (# mode::<
          (#  do  LCD.display_minutes;  Set_minutes##  ->  this(Watch)._STATE
#);
         inc::< (# do LCD.inc_hours #)
      #);
     Set_Minutes: STATE
      (#  mode::<  (#  do  LCD.display_time;  Display##  ->  this(Watch)._STATE
#);
         inc::< (# do LCD.inc_minutes #)
      #)
  #)
```

Figure 7

5. Implementation

Since the days of Simula, a common technique to implement virtual functions is to use a virtual dispatch table (VDT) and all calls of virtual functions are made via the dispatch table. In this way it is the actual object denoted by a reference that determines which virtual function is executed.

The state mechanism may be implemented by having an additional virtual dispatch table for each state. A state change can then be implemented by changing the VDT reference of the object. For the Watch example, the implementation is illustrated in Figure 8a. Here is shown an instance of class Watch (aWatch object) in three different states (L1, L2, L3). In addition three VDTs are shown, one corresponding to the class Watch and one for each of the internal state patterns. There is in fact a fourth VDT corresponding to the standard VDT for class Watch, but this VDT is not shown in the figure. Initially (at L1) the VDT is set to refer to the VDT for the state Display. This initialization happens when the procedure Init is executed. The VDTs for Display, Set_hours and Set_minutes do also have an entry corresponding to Init, but this entry is not shown in the Figure.

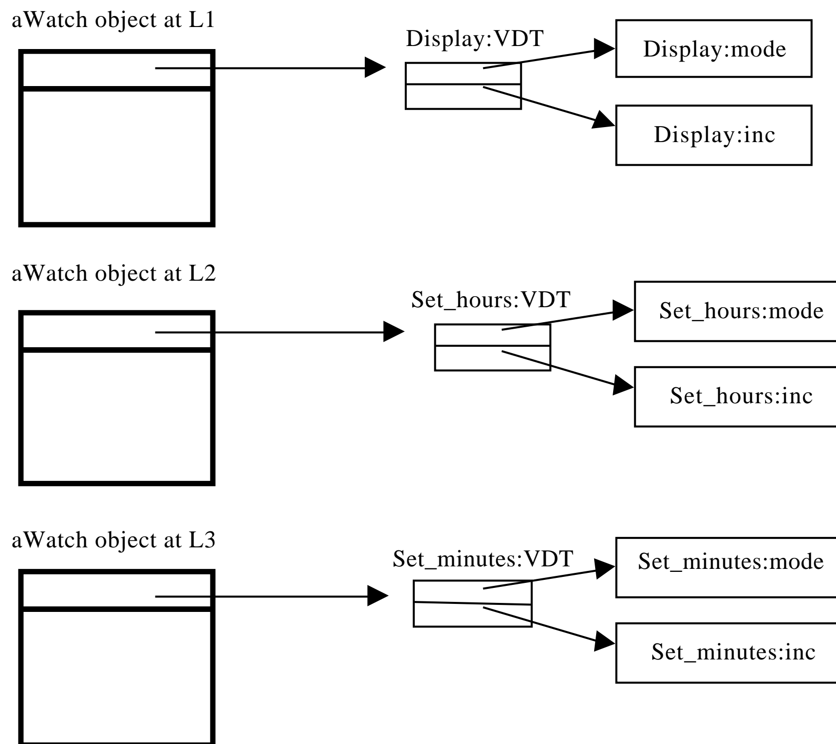

Figure 8a

In Figure 8b, its is shown how the VDT of `aWatch` object may be changed by assigning it a new state corresponding to the labels `L1`, `L2` and `L3` in Figure 8a. As can be seen, the implementation of the state concept is a simple assignment to the VDT field.

```
Display## -> this(Watch)._STATE;    (* L1 *)
...
Set_hours## -> this(Watch)._STATE   (* L2 *)
...
Set_minutes## -> this(Watch)._STATE(* L3 *)
...
Display## -> this(Watch)._STATE     (* L1 *)
...
```

Figure 8b

6. Inheritance and state machines

When using state machines together with object-orientation, it should be considered how state machines could be used in combination with inheritance from superclasses. Here we show two different techniques for using inheritance together with subpatterns. First we show how to use inheritance to describe nested states. Then we show how to define a subclass of a given class and extend the states of the superclass.

6.1 Nested states

Statecharts may be nested in the sense that a state may have internal states. Consider a blower with 3 buttons, on, off, and step. The blower may run with 2 different speeds. When turned on it starts running with a low speed. If the step button is pushed, it changes to a higher speed. The step button will then alternate the blower between the lower and the higher speed. No matter which speed the blower is running, the off button will stop the blower. The operations of the blower are illustrated in Figure 9.

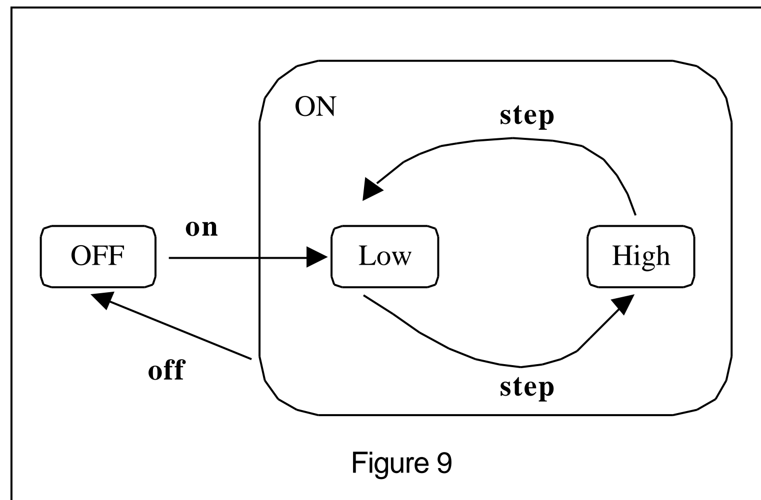

Figure 9

As can be seen the state ON has two substates, Low and High. Pushing the off button will make a transition to state OFF independent of whether being in substate Low or High. Nested states may be modeled by means of inheritance on states. The Blower may be described in BETA as shown in Figure 10.

```
Blower:
  (# switchOn:< (# do inner #);
     switchOff:< (# do inner #);
     step:< (# do inner #);
     init:< (# do OFF## -> this(Blower)._STATE #);
     OFF: STATE
       (# switchOn::< (# do LOW## -> this(Blower)._STATE #)
       #);
     ON: STATE
        (# switchOff::< (#do OFF## -> this(Blower)._STATE #);
     LOW: ON
       (# step::< (#do HIGH## -> this(Blower)._STATE #)
       #);
     HIGH: ON
       (# step::< (#do LOW## -> this(Blower)._STATE #)
       #);
  #)
```

Figure 10

The states LOW and HIGH have a common superstate, ON, which defines a common binding of the virtual pattern switchOff.

6.2 Inheriting and Extending State Machines.

One of the major advantages of OO languages is the ability to define a general superclass covering the general properties of a given set of objects. More specific objects may then be defined using subclasses. A successful integration of OO and state machines should make it possible to define general state machines and refine these state machines in subclasses. In the BETA context this can be done by means of virtual states.

In Figure 11a, a sketch of a class A with virtual states is shown. The class pattern A is similar to the class pattern P in Figure 6, except that the states S1 and S2 are declared virtual. In Figure 11b, a subpattern AA of A is defined. In AA two new virtual functions g1 and g2 are added, the states S1 and S2 are extended and a new state S3 is defined. To be

```
A: (# f1:< ...;
      f2:< ...;
      f3:< ...;
      S1:< STATE
           (# f1::< ...;
              f2::< ...;
              f3::< ...;
           #);
      S2:< STATE
           (# f1::< ...;
              f2::< ...;
              f3::< ...;
           #);
      ...
   #)
```

Figure 11a

```
AA: A
 (# g1:< ...;
    g2:< ...;
    S1::<
      (# f1::< ...;
         g1::< ...;
         g2::< ...;
      #);
    S2::<
      (# f2::< ...;
         g2::< ...;
      #);
    S3:< STATE
      (# f3::< ...;
         g2::< ...
      #)
 #)
```

Figure 11b

more specific:

- Class pattern AA is a subpattern of A.
- AA inherits the virtual functions f1, f2 and f3 and the virtual states S1 and S2 in the usual way
- AA defines two new virtual functions g1 and g2.
- AA extends the states S1 and S2 inherited from A.
- The extension of S1::< ... extends the definition of f1 from S1 in A. This is a normal BETA further binding of a virtual function. In addition S1 defines bindings of g1 and g2.
- The extension of S2 makes a further binding of f2 and specifies a binding of g2.
- Finally a new virtual state S3 is added. S3 specifies bindings of f3, and g2.

6.2.1 Example

An example of a general state class is shown in Figure 12a. It defines a class Unit which may be in one of the states OFF or ON. Class Unit describes the general properties of Units that may be turned on and off including the transitions between these two states. The states OFF and ON are virtual and may thus be extended in subpatterns of Unit. In Figure 12b is shown a subclass CDplayer of class Unit. Class CDplayer extends Unit in the following way:

Two new virtual functions play and stop are added.

```
Unit:
(# switchOn:< ...;
   switchOff:< ...;
   ON:< STATE
    (# switchOff::<
         (#
         do OFF## ->
            this(Unit)._STATE
         #);
    #);
   OFF:< STATE
    (# switchOn::<
         (#
         do ON## ->
            this(Unit)._STATE
         #)
    #)
#)
```

Figure 12a

```
CDplayer: Unit
(# play:< ...;
   stop:< ...;
   ON::<
    (# play::<
         (#
         do Playing## ->
            this(CDplayer)._STATE
         #)
    #);
   Playing:< STATE
    (# stop::<
         (#
         do ON##->
            this(CDplayer)._STATE
         #)
    #)
#)
```

Figure 12b

- A new state `Playing` is added.
- The state `ON` is extended to have a binding of the new virtual function play. Execution of play makes a transition into the new state `Playing`.
- State `Playing` defines a binding of stop, which brings the `CDplayer` into state `ON`.

In Figure 13 the state machines for `Unit` and `CDplayer` are illustrated. The dashed lines indicate that the states OFF and ON are inherited from `Unit`.

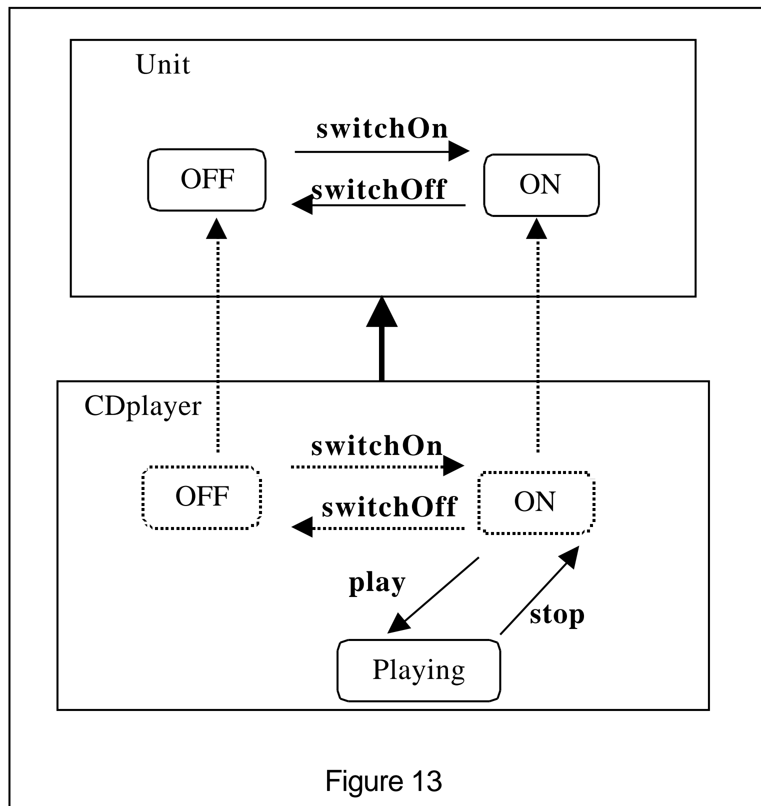

Figure 13

6.2.2 Virtual states and virtual class patterns

In BETA parameterized classes are defined by means of virtual class patterns, which is a simple generalization of virtual functions. It is outside the scope of this paper to discuss virtual classes. For details see [MM89, MMM90, MMN93].

The concept of virtual states as described above, is very similar to virtual class patterns. The form used here for virtual states corresponds to defining nested virtual class patterns.

7. Related work and conclusion

The idea of combining classes and state machines in the simple style adapted for BETA first originated in [Tai93] where it was proposed to be able to change the dispatch pointer for an object at run-time. This is a very simple and elegant idea, which deserves further investigation and we hope that the BETA implementation contributes to this.

When developing software for embedded systems, memory space is often limited. When using the state pattern, there will be an object corresponding to each state and each object has an associated dispatch table. Using the proposed mechanism, the extra state objects are avoided and only the dispatch tables are present. This is of course a minor difference, but may be important if there are many states and limited memory.

Changing the dispatch pointer may be seen as a simple case of changing class membership of an object. Early dynamic OO languages like Smalltalk and CLOS have operations for doing this and the class of an object may be changed to an arbitrary other class.

In the prototype-based language Self [US88], it is possible to obtain a similar effect by means of dynamic parent pointers. The experience from Self [BUW99] is, however, that dynamic parent pointers easily leads to complicated programs that are hard to debug. It remains to be seen whether or not the state mechanism proposed in this paper has similar problems.

In a structured world with static typing, there are number of problems in changing the class of an object. There are many reasons to be able to change the class of an object dynamically, since the properties of an object may depend on its actual state.

Some of the more general work on dynamic classification of objects is the work on predicate classes [Cha93,EKC98]. With predicate classes it is possible to define a class by means of a predicate over the instance variables of an object. This is a very general mechanism and it is clearly much more expressive than the state classes of [Tai93] and their implementation in BETA. As a language mechanism, predicate classes are complex and can hardly be characterized as a primitive language mechanism. A state class is on the other hand a simple mechanism and as mentioned above, it may be implemented by a change of the dispatch pointer. Type checking of predicate classes is also a non-simple issue. For state classes, type checking is simple. For each state it must be checked that the virtual functions are in fact defined as virtuals in the enclosing class.

The main contributions of this paper are:

- Present an integration and implementation of state classes in a real programming language.
- Present mechanisms for combining state machines and inheritance to represent nested states and inheritance for state machines.

A main motivation for this work was to contribute to a further integration of programming languages and graphical design notations. As said in the introduction, one of the advantages of object-orientation is that it provides an integrating perspective on analysis, design and implementation. Main-stream OO methodologies makes use of graphical design languages like UML and programming languages like C++, and Java and the use of different languages give unnecessary complications.

A CASE tool component for supporting state machines and state classes with incremental code generation and reverse engineering is being constructed. As there is a direct correspondence between state machines and state classes it is straightforward to construct such a tool. As for the Mjølner CASE tool where the diagrams and the code are just two different views of the underlying abstract syntax tree, state machines or state classes will also "just" be different views of the same underlying abstract syntax tree.

Acknowledgements. This work has been carried out as part of *Centre for Object Technology (COT)*, which is a joint research project between Danish industry and universities (`http://www.cit.dk/COT`). COT is sponsored by The Danish National Centre for IT Research (CIT), the Danish Ministry of Industry and Aarhus University. The ideas presented in this paper emerged during work in COT case 2. The author wants to thank

the participants of case 2 for useful discussions and Carsten Bjerring for commenting on an earlier version of this paper. Also thanks to Lars Bak, Erik Ernst, Antero Taivalsaari, Dave Ungar, and Mario Wolczko, for interesting discussions at Sun Labs on the subject of this paper.

8. References

[AG98] K. Arnold, J. Gosling: The Java Programming Language Second Edition, Addison Wesley, 1998.

[AL99] C.J. Andersen, M.L. Lauridsen: A Tool for Generating Sequence Diagrams, Center for Object Technology, Technological Institute, Bang & Olufsen, Working Note in progress, 1999.

[BUW99] L. Bak, D. Ungar, M. Wolczko: Personal communication, 1999.

[Boo91] G. Booch: Object-Oriented Design with Applications, Benjamin Cummings, 1991.

[BRK99] G. Booch, J. Rumbaugh, I. Jacobsen: The Unified Modeling Language User Guide, Addison Wesley, 1999.

[Cha93] C. Chambers: Predicate Classes, European Conference on Object-Oriented Programming, ECOOP'93, pages 268 - 296, LNCS-707, Springer Verlag 1993.

[DNM68] O.J. Dahl, K. Nygaard, B. Myrhaug: Simula 67 Common Base Language, Technical Report, Publ. no. S-2, Norwegian Computing Center, Oslo, 1968.

[EKC98] M. Ernst, C. Kaplan, C. Chambers: Predicate Dispatching: A Unified Theory of Dispatch, European Conference on Object-Oriented Programming, ECOOP'98, pages 186 - 211, LNCS-1445, Springer Verlag 1998.

[EP98] H.-E. Eriksson, M. Penker: UML Toolkit, John Wiley & Sons, New York, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[GR89] A. Goldberg, D. Robson: Smalltalk-80, the Language and its Implementation, Addison Wesley, 1989.

[Har87] D. Harel: Statecharts: A Visual Formalism for Complex Systems, Sci. Computer Prog., July 1987, pp. 231-274.

[Kee89] S.E. Keene: Object-Oriented Programming in COMMON LISP - A Programmers Guide to CLOS, Reading MA: Addison Wesley, 1989.

[KLMM94] J.L. Knudsen, M. Löfgren, O.L. Madsen, B. Magnusson: Object-Oriented Environments – The Mjølner Approach, Prentice Hall, 1994.

[MM89] O.L Madsen, B. Møller-Pedersen: Virtual Classes, a Powerful Mechanism in Object-Oriented Languages, Conference on Object-Oriented Programming, Languages, Systems and Applications, OOPSLA'89, ACM Sigplan Notices, **24**(10), 1989.

[MMM90] O.L. Madsen, B. Magnusson, B. Møller-Pedersen: Strong Typing of Object-Oriented Languages Revisited, Conference on Object-Oriented Programming, Languages, Systems and Applications, OOPSLA'90, ACM Sigplan Notices, **25**(10), 1990, also in [KLMM94].

[MMN93] O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Object-Oriented Programming with the BETA Programming Language, Addison Wesley/ACM Press, 1993.

[Mad94] O.L. Madsen: An Overview of BETA, in [KLMM94].

[Mad98] O.L. Madsen: BETA: A Block-Structured Concurrent Object-Oriented Language, *in Handbook of Object-Technology (S: Zamir, ed.), CRC Press, 1998.*

[Str86] B. Stroustrup: The C++ Programming Language, Addison-Wesley, 1986.

[Tai93] A. Taivalsaari: Object-Oriented Programming with Modes, Journal of Object-Oriented Programming, pages 25-32, June 1993.

[US87] D. Ungar, R. Smith: Self, the Power of Simplicity, Conference on Object-Oriented Programming, Languages, Systems and Applications, OOPSLA'87, ACM Sigplan Notices, **22**(12), 1987.

[Zül98] H. Züllighoven, Das Objekorienterte Konstruktionshandbuch, dpunkt.verlag, 1998.