

# Object-Oriented State Machines: Subclassing, Composition, Delegation, and Genericity

Aamod Sane\* and Roy Campbell

University of Illinois at Urbana-Champaign  
Department of Computer Science  
1304 W. Springfield Avenue, Urbana, IL 61801  
email: {sane,roy}@cs.uiuc.edu  
www: <http://choices.cs.uiuc.edu/sane/home.html>

## Abstract

Software specification and implementation techniques based on state machines simplify design, coding, and validation. However, large systems require complex state machines. Incremental construction techniques can control this complexity. In this paper, we present a construction technique that permits derivation of complex state machines from simpler state machines. The technique uses subclassing, composition, delegation, and genericity to incrementally modify and combine simpler machines.

In addition, we present a novel implementation technique that uses exactly one table-lookup and one addition to dispatch events on derived state machines, no matter the depth of the derivation. As an example, we describe the derivation of a complicated distributed virtual memory scheme from a simple paging virtual memory scheme.

## 1 Introduction

Many designers advocate the use of state machines to specify and implement software systems. For example, reactive system designs [10] and object-oriented analysis and design [26, 2, 5] use state machines. However, large systems require complex state machines. In this paper, we introduce object-oriented techniques that permit the design of complex state

machines by incrementally modifying and combining other independently designed machines. In addition, we present a novel, highly efficient technique for implementing state machines that extends object-oriented approaches such as the *reification technique* (RT) [13, 8]. Our technique facilitates incremental derivation of state machines using subclassing, composition, delegation, and genericity (henceforth called object-oriented-techniques (OOT)), and improves upon the reuse afforded by RT.

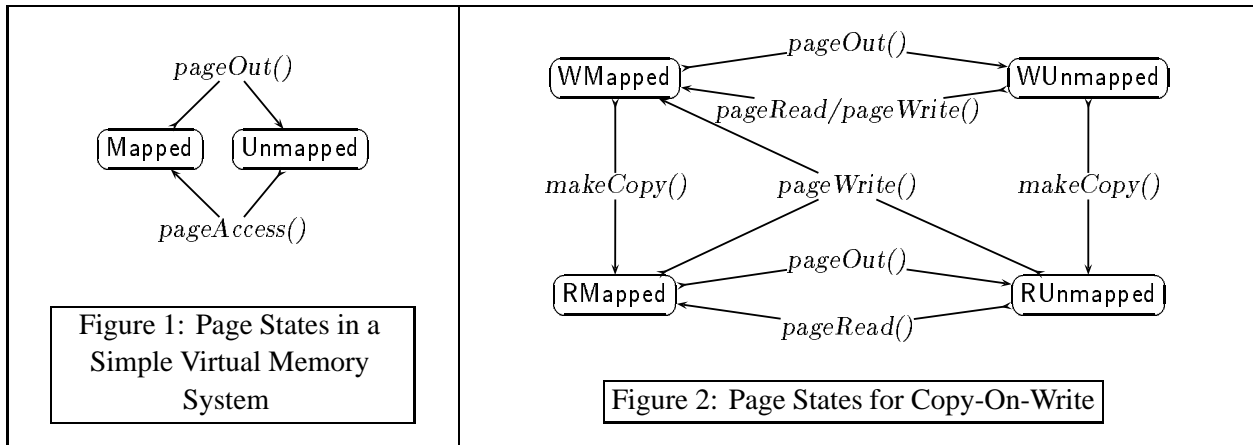
We motivate the use of subclassing by considering a simple virtual memory (VM) paging scheme (Figure 1) that is enhanced to support copy-on-write (COW) [1] (Figure 2). In Figure 1, a virtual memory page may be mapped into physical memory so that it is accessible. The page may be unmapped to store it on backing store and release physical memory for other use<sup>1</sup>. The state machine in Figure 2 supports copy-on-write. COW allows data created by one process to be shared with a different process without requiring the data to be copied. Instead, the physical pages on which the data resides are shared between processes until the processes modify them. Data is “copied” by mapping the associated physical page into the virtual address space of the target process with read-only access. However, upon a write access, the “copied” data is duplicated by copying the page to a new physical page and changing the read-only access to write access. The two figures show several similarities, for example states *RMapped*, *WMapped*, are simi-

---

\*Supported by CNRI contract CNRI GIGABIT/UIILL.

---

<sup>1</sup>For simplicity, the state machine diagrams do not show loops (transitions that do not change state).



lar to `Mapped` and methods `pageRead()`, `pageWrite()` are similar to `pageAccess()`. The two figures have differences corresponding to the additional behavior, for example, `makeCopy()` is added and `pageWrite()` causes transitions from `RMapped` to `WMapped`.

Using the techniques in this paper, the similarities between the two state machines can be captured by *subclassing*. VM machine code corresponding to similar transitions can be reused in the COW machine, and new COW code need be written only for additional behavior. The paper presents other examples showing how complex state machines that support *distributed virtual memory* (DVM) are also easily derived using OOT, while maximizing reuse.

Besides maximizing reuse, our techniques also show that in order to properly define subclassing, composition, etc., for state machines, we have to reconsider the usual notion of *Self* (“this” in C++). Section 2 considers this issue in greater detail. In the remainder of the paper<sup>2</sup>, we give examples of composition, delegation, and genericity in Section 3. The definition of *Self* (Section 2) leads to formal definitions of OOT in terms of *embeddings* (Section 4). Analysis based on embeddings suggests a fast implementation involving a *single indexed table lookup* (Section 5), regardless of the height of the subclassing, composition, or delegation hierarchy. A space-efficient implementation of generics requires one more lookup, but the lookup

<sup>2</sup>We use Mealy machines [11], but since actions may be associated with either source or destination state objects, our technique also applies to Moore machines [11].

can be eliminated if necessary. Section 6 summarizes topics such as subtyping and nested states that are not discussed in detail due to space restrictions. We consider related work in Section 7, and in the conclusion (Section 8) we indicate possible generalizations of our techniques and other avenues for further research.

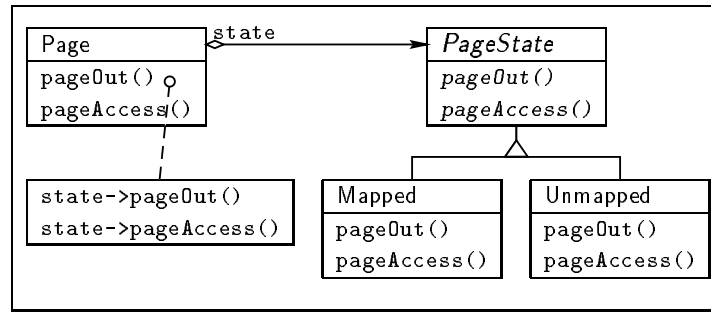
## 2 Subclassing

In this section, we consider how to derive the COW machine (Figure 2) as a subclass of the Simple VM machine (Figure 1). We first show that standard techniques such as RT [13, 8] block certain kinds of reuse. Then we observe that this is *not* an artifact of any particular implementation technique. Reusing code while subclassing state machines is difficult because the IS-A relationship between states of base and derived machines, and the BECOMES-A relationship induced by state transitions, interact with each other. Hence, a general solution requires a redefinition of the notion of *Self* that accounts for both relationships.

**Reuse restrictions in RT** Figure 3 shows an implementation (in OMT-like notation [21]) of the VM state machine (Figure 1) using RT<sup>3</sup>, where states are reified as objects, and method calls implement state transitions.

In Figure 3, pages are instances of the class `Page`. `PageState` is the abstract class that declares the methods for each state of a page, while the classes

<sup>3</sup>We build on RT. For comparisons between RT and other techniques see [8].



Legend:  $\triangleleft$  inheritance,  $\diamond$  instance variables,  $\circ$  method implementation.

Figure 3: Reifying States as Objects

Mapped and Unmapped represent the concrete states of a page. Each page object maintains a current state variable *state*, that points to an instance of Mapped or Unmapped. Methods of class *Page* (e.g., *Page::pageOut()*) invoke the corresponding method of the *PageState*. For instance

```
Page::pageOut()
{ state = state->pageOut(this); }
```

The state methods define actual behavior, as in

```
Mapped::pageOut(Page * p)
{ p->flushMMU(); p->writeToDisk();
  return Unmapped::Instance(); }
```

The state objects themselves have no instance variables, so there is only one systemwide instance of each concrete state. The method call *Unmapped::Instance()* returns a pointer to the system-wide instance.

Using RT, the COW machine might be implemented directly in a similar manner. However, suppose that we want to reuse VM machine code for methods such as *Mapped::pageOut()*, since the behavior is similar to that of *WMapped::pageOut()* and *RMapped::pageOut()*. We might derive a class *WMapped* from the class *Mapped*, hoping to reuse *Mapped::pageOut()*. But now there is a problem: where *Mapped::pageOut()* returns *Unmapped::Instance()*, *WMapped::pageOut()* must return *WUnmapped::Instance()*. Although behavior in the two states is similar, the *state transitions* differ for the COW machine. If we attempt to reuse *Mapped::pageOut()* by redefining *Unmapped::Instance()* to return *WUnmapped::Instance()*, we find that *RMapped::pageOut()* cannot reuse *Mapped::pageOut()*, as it must return *RUnmapped::Instan-*

*ce()*. Thus, with RT, the code for *Mapped::pageOut()* cannot be reused in both methods without modification.

In abstract terms, the problem is that since *pageOut()* is implemented in the superclass, it cannot distinguish between the states *WMapped* and *RMapped*. We observe that when state machines are derived, one transition in the base state machine can correspond to a number of transitions in the derived states. In general, a transition's source and destination state arities may change in unrelated ways during subclassing. Therefore, we need two different ways to map the source and destination states of a transition. In the following, we present an implementation based on this observation.

**Solution using StateMaps** Instead of returning a programmed constant, we return the next state *indirectly* by looking-up the actual state we want to return in tables called *StateMaps*. Derived machines are then constructed by systematic modification of *StateMaps*, and base state methods are inherited without change.

A *StateMap* is similar to a C++ *VTable* [7]: C++ *virtual functions* are dispatched using the *VTable*, so that derived classes can replace “virtual functions” of the base class with their own versions. Similarly, a *StateMap* is used to determine the actual *next* state, so that a derived state class may “replace” it with the next state appropriate for the derived machine. Furthermore, just as every C++ class has its own *VTables*, every state has its own *StateMap*.

Using StateMaps (Figure 4), we write:

```
Mapped::pageOut(Page * p)
{ p->flushMMU(); p->writeToDisk();
return map->Unmapped(); }
```

where `map` is an instance of `StateMap`. Method `map->Unmapped()` invokes `Unmapped::Instance()`, returning the pointer to the system-wide instance of `Unmapped`.

Figure 5 shows how the COW machine states are now derived from VM machine states. Class `WMapped` is derived from class `Mapped`, but instances of `WMapped` are initialized to have pointers to an instance of class `StateMapW`, which is itself derived from `StateMap`. Now if `pageOut()` is actually invoked from an instance of `WMapped`, `pageOut()` will invoke `map->Unmapped()`. Since in `WMapped`, “`map`” is an instance of `StateMapW`, `map->Unmapped()` returns `WUnmapped::Instance()` as required. Similarly, `RMapped` may *independently* redefine `map->Unmapped()` using `StateMapR`.

**Self, Becomes-A, Is-A, and Embeddings** In usual object-oriented programming, the *Self* (“this”) of an object may be regarded as a function that binds method invocations at run-time. We have seen that for state machines, we need two functions, one for the source and other for the destination state. This is because upon subclassing state machines, the IS-A relation defined by inheritance between state objects, and the BECOMES-A relation defined by state transitions *interact*. For instance, in Figure 5, the relation “a `WMapped` page IS-A `Mapped` page” is programmed using ordinary inheritance between the immutable state objects, while the relation “a `Mapped` page BECOMES-A `Unmapped` page upon `pageOut()`” is incrementally modified to “`WMapped` page BECOMES-A `WUnmapped` page upon `pageOut()`” using `StateMaps`. Thus, for state machines, *Self* is a *pair* of functions that define how a transition in the base machine is mapped into a transition in a derived machine. We say that the base machine is *embedded* into a derived machine. Formal definitions based on this idea are presented in Section 4. But first, we see some examples of composition, delegation, and genericity.

### 3 Composition, Delegation, and Genericity

#### 3.1 Composition

State machines are composed to combine behaviors defined in component machines. We demonstrate composition by constructing a *distributed virtual memory* (DVM) protocol machine (Figure 8) out of a virtual memory machine (Figure 6) and a networking machine (Figure 7). DVM [24] provides the illusion of a global shared address space over networks of workstations, whose local memories are used as “caches” of the global address space. The caches have to be kept consistent: a simple approach allows only one machine to access a shared page at a time. If another machine attempts to access that page, its virtual memory hardware intercepts the access, and its fault handler fetches the page from the current page-owner. Thus, behavior for DVM has VM and networking aspects. We define VM and networking behavior using separate state machines, and compose them to get a DVM machine.

**DVM-VM MACHINE:** In a DVM system, pages may be either `DMapped`, `DUnmapped` or `Remote` (Figure 6). The `DMapped` and `DUnmapped` states are inherited from the simple VM machine (Figure 1). State `Remote` represents a page on some remote machine, and `remAccess()` defines actions for pages accessed by a remote machine. The transitions are defined as though no networking were necessary. (The special state `Null` ignores all VM actions; it is used in composition. We always create `Null` and `Error` states for every state machine).

**DVM-NET MACHINE:** The networking machine (Figure 7) implements a trivial protocol that sends a page to a remote machine, or gets one from a remote machine. It handles details like fragmentation and sequence numbering.

**DVM MACHINE:** In the composite `DVMMachine` (Figure 8), suffix `Q` indicates that in the composite state, the DVM-Net state is `Quiescent`, and suffix `N` indicates that the VM state is `Null`. We implement transitions to and from `RemoteQ` using the networking machine. `MappedQ` and `UnmappedQ` inherit behavior from the DVM-VM machine.

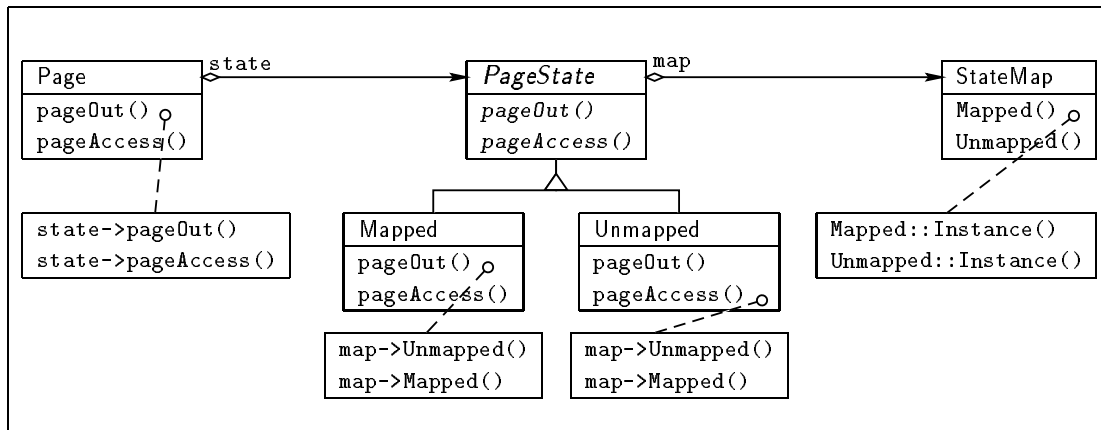
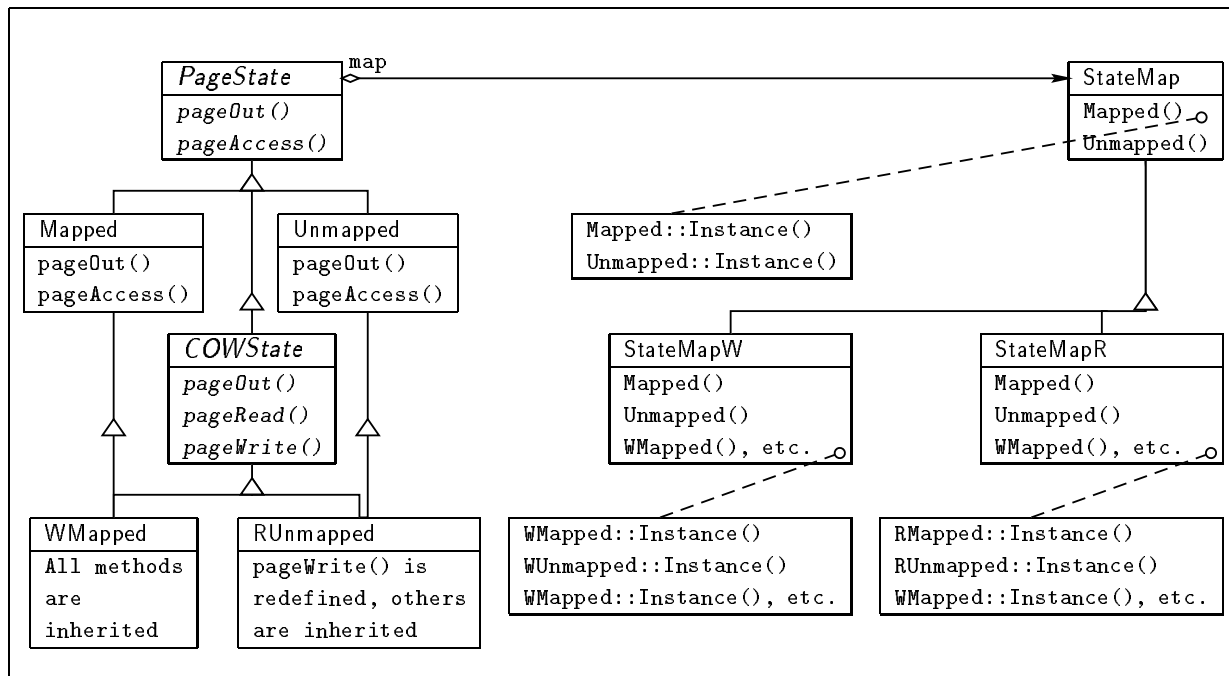


Figure 4: VM Machine Using StateMaps



Class Page (see Figure 4), state classes WUnmapped and RMapped, and *makeCopy()* were omitted for brevity. *pageRead()* and *pageWrite()* inherit behavior from *pageAccess()*, but for RMapped and RUnmapped, *pageWrite()* also makes physical copies and changes protection. *makeCopy()* has to be defined for WMapped and WUnmapped. Otherwise, no new code is necessary.

Figure 5: Subclassing VM machine for COW using StateMaps

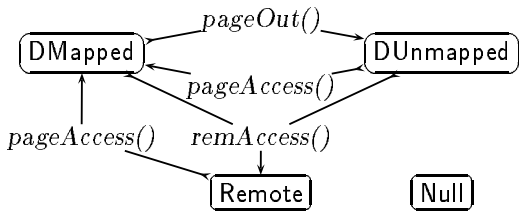


Figure 6: DVM-VM State Machine

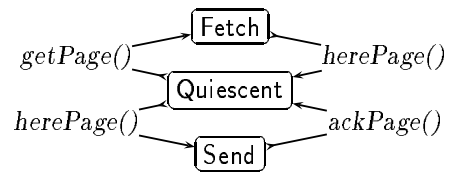


Figure 7: DVM-NET State Machine

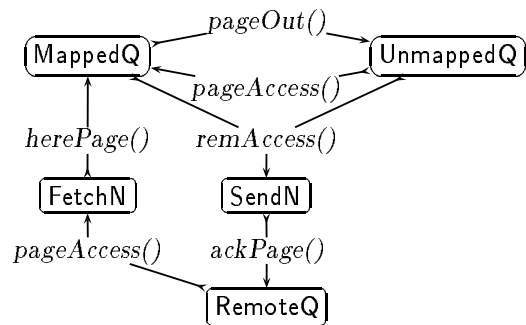


Figure 8: DVM Composite State Machine

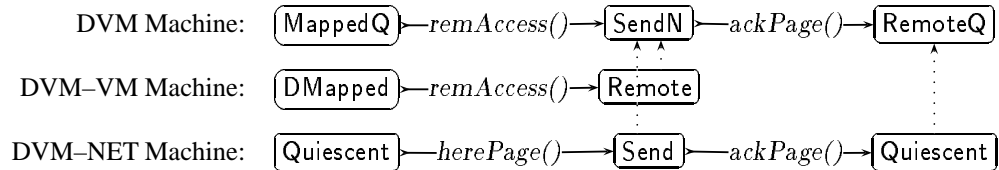


Figure 9: Example of Composition

Methods of the composite machine reuse behavior defined in methods of component machines by initializing StateMaps so that the states returned from component methods are actually composite states. For instance, Figure 9 shows how the composite method `MappedQ::remAccess()` combines and reuses `DMApped::remAccess()` and `Quiescent::herePage()`. In the DVM-NET machine, `Quiescent::herePage()` returns the `NET` state `Send`, but when invoked from the composite state `MappedQ`, it returns the composite state `SendN`. In turn, `ackPage()` when invoked from `SendN`, returns `RemoteQ` instead of `Quiescent`. `RemoteQ` later gets used as a VM state.

### 3.2 Delegation

Delegation can be programmed using state machines [13], and in turn state machines may delegate behavior to other machines. For instance, Figure 3 might be considered a description of how instances of `Page` delegate their behavior to the “page classes” `MappedPage` and `UnmappedPage` (see also [20]). The class of a `Page` object changes when it changes states, effectively changing the delegatee. Similarly, state machines may “delegate” their behavior to other state machines, and the delegatees can be changed at run-time, much as individual states are changed in ordinary state machines.

As an example, consider the COW machine (Figure 2). When a page has no copies (states `WMapped` and `WUnmapped`), the COW machine behaves like a vanilla VM machine (Figure 1). However, when a page is copy-on-write, the COW machine behaves like a restricted version of the VM machine that allows only `pageRead()` — call this the RVM machine. With this viewpoint, a COW machine may be regarded as a two-state machine, with states `CMapped` and `CUnmapped`, that delegates its events to the state machines `VM` and `RVM` (Figure 10). The methods `pageWrite()` and `makeCopy()` changes the delegatees, changing the behavior of every state *at the same time*.

### 3.3 Genericity

We say that a state machine is a *template* if it declares states and events, and its actions are parametr-

ized by states (and possibly other parameters). Such a template is instantiated binding parameters at compile time. Templates themselves can be subclassed and composed; conversely, subclassed machines may be used to instantiate the same template. Thus, genericity can lead to interesting patterns of reuse. For example, consider a DVM machine that allows multiple copies of read-only pages. Such a DVM-COPY machine is itself composed from a DVM-VM-COPY machine, and a DVM-NET-MULTICAST machine. These new components can be themselves derived from the DVM-VM and DVM-NET machines respectively, and are also composed in similar ways.

We take advantage of this by defining templates for DVM-VM (Figure 6) and DVM-NET (Figure 7) machines, and composing the templates to create a template for the DVM machine (Figure 8). Now we can subclass this template to define the composition of more complex components. It is then instantiated with state machines for DVM-VM-COPY and DVM-NET-MULTICAST. Figure 11 shows such a construction.

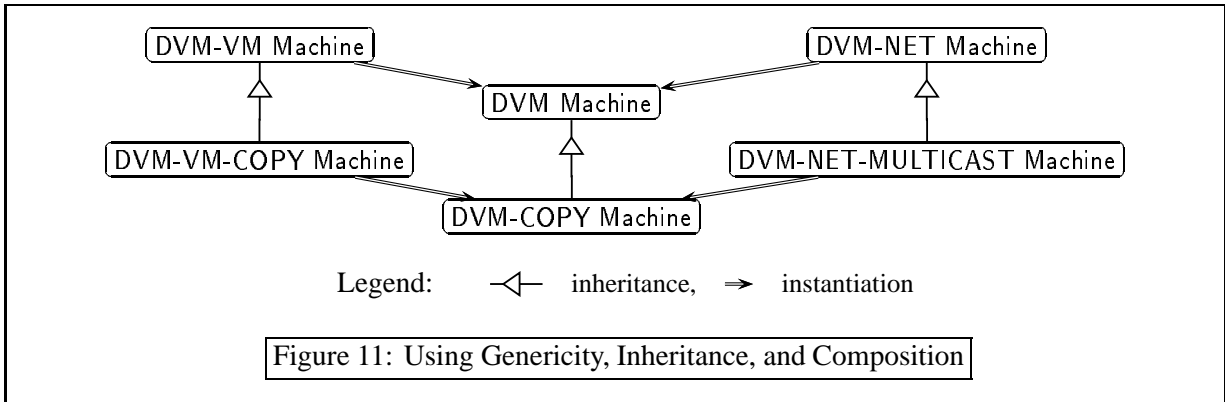
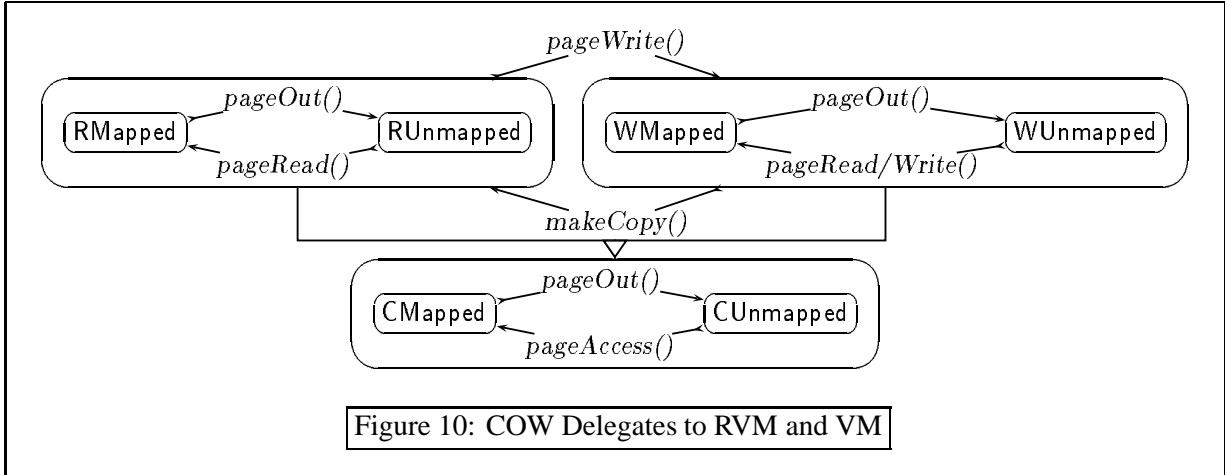
Section 5 describes implementation techniques for all the constructions above. However, in order to understand the implementation, we need precise definitions of our four constructions.

## 4 Embedding

Sections 2 and 3 give examples in which machines are derived by simply modifying StateMaps. In this section, we make the notion of modification precise using *embeddings* to define the four OOT’s.

In Section 2, we observed that state machine code can be inherited if the *current state*, the *next state* and the *action method* are bound dynamically. Therefore, we define subclassing, composition, etc., in terms of mappings between triples of *source state*, *event*, *destination state*. Such triples are called *transitions*, and mappings between transitions are called *embeddings*.

More formally, using the standard [11] definition of state machines, define two state machines  $\mathcal{B}$  (base)



and  $\mathcal{D}$  (derived):

$$\begin{aligned} \mathcal{B} & \langle States_b, Events_b, Trans_b \rangle \\ Trans_b & \subseteq (States_b \times Events_b \times States_b) \end{aligned}$$

$$\begin{aligned} \mathcal{D} & \langle States_d, Events_d, Trans_d \rangle \\ Trans_d & \subseteq (States_d \times Events_d \times States_d) \end{aligned}$$

All state and event sets are disjoint.

A state machine has states, events and transitions. Transitions are triples such as  $\langle S, ev(), D \rangle$ , where  $S$  is the source state,  $ev()$  the event, and  $D$  the destination state. An event need not have transitions to all possible destinations, so  $Trans_b$  and  $Trans_d$  are subsets of the cartesian product. But note that all combinations of source state and event must occur in the transition relation. We do not distinguish between input events and output actions. In object-oriented state machines, the *name* of a method is the input event, and its *definition* constitutes the output action. Methods of a state object can be invoked either when it is the current state, the next state, or both, so both the Moore and Mealy machines are easily implemented.

Embeddings are defined as mappings from one set of transitions to another, e.g., from  $Trans_b$  to  $Trans_d$ . Embeddings need not be functions, since one transition in a machine can be mapped to many transitions in another machine. In the following, we define subclassing, composition, delegation and genericity using embeddings.

#### 4.1 Subclassing, Composition, Delegation, and Genericity

The OO constructions are defined as embeddings that satisfy certain consistency conditions. Note that these definitions do not guarantee *subtyping* (the definitions specify the *mechanism*), since subtyping is a behavioral notion (subtyping specifies the *policy*). Section 6 considers subtyping.

**Definition of Subclassing** We say that a state machine  $\mathcal{D}$  is derived by *subclassing*  $\mathcal{B}$ , if there is an *embedding* (i.e., a mapping) from the set  $Trans_b$ , the set of transitions of  $\mathcal{B}$ , to the set  $Trans_d$ , the set of transitions of  $\mathcal{D}$ , such that:

1. A transition  $t_b$  of the state machine  $\mathcal{B}$  ( $t_b \in Trans_b$ ) may be mapped to one or more transition  $t_d$  of the state machine  $\mathcal{D}$  ( $t_d \in Trans_d$ ).
2. For every transition  $t_d$  in the derived machine  $\mathcal{D}$ , either there is exactly one transition  $t_b$  in the base machine  $\mathcal{B}$  such that  $t_b$  is mapped to  $t_d$ , or the event for the transition  $t_d$  is new to  $\mathcal{D}$ .

The first condition allows many states in a derived machine to inherit behavior from one state in the base machine. The second condition ensures that every machine in the derived state defines some behavior (possibly a null or error state) for every event defined in the base machine. The second condition also implies that no two transitions  $t_b$  and  $u_b$  from  $Trans_b$  are mapped to the same transition  $t_d$  from  $Trans_d$ , so derived states do not have ambiguous behavior.

As an example, consider the VM (Figure 1) and COW (Figure 2) state machines from Section 2. The transition  $\langle \text{Unmapped}, \text{pageAccess}(), \text{Mapped} \rangle$  from the VM machine is mapped to  $\langle \text{WUnmapped}, \text{pageRead}(), \text{WMapped} \rangle$  and to  $\langle \text{RUnmapped}, \text{pageRead}(), \text{RMapped} \rangle$ , as well as the transitions for  $\text{pageWrite}()$ . The embedding from VM to COW satisfies all of the conditions, so the COW machine is a subclass of the VM machine.

When a transition from a base machine is mapped to a derived machine, in the scheme of Figure 5, for each transition the mapping is programmed as follows:

- The class for the *source* state in the derived machine inherits from the class of the source state in the base machine. In our example, the classes `WUnmapped` and `RUnmapped` are both subclasses of `Unmapped`.
- The *destination* state in the transition from the derived machine “replaces” the destination state of the base machine. This replacement is programmed by using different `StateMaps`, as we discussed in Section 2. The destination states need not be subclasses.
- An *event* (method) of the base machine is mapped using virtual functions or equivalent mechanism in the language, or by programming an actual call. Thus, in Figure 5, the `pageOut()` method is inherited in the COW states as a virtual function,

however, the `pageRead()` and `pageWrite()` methods actually invoke the `pageAccess()` method.

This method works for many cases, however, in Section 5 we see it does not always suffice.

**Definition of Composition** We say that a state machine  $\mathcal{D}$  is derived by *composing* two (or more) machines  $\mathcal{B}$  and  $\mathcal{C}$  if there is an *embedding* (i.e., a mapping) from the sets  $Trans_b$  and  $Trans_c$  to the set  $Trans_d$ , such that both maps from  $Trans_b \rightarrow Trans_d$  and  $Trans_c \rightarrow Trans_d$  individually satisfy the conditions for subclassing, but we need to replace condition 2: For every transition  $t_d$  in the derived machine  $\mathcal{D}$ , either there is a transition  $t$  in at least one of  $\mathcal{B}$  or  $\mathcal{C}$  such that  $t$  is mapped to  $t_d$ , or the event for the transition  $t_d$  is new to  $\mathcal{D}$ <sup>4</sup>.

Examples of transitions for composite machines are readily apparent in Figures 6, 7, and 8. Furthermore, since each mapping is individually a subclass mapping, maps between transitions may be programmed in the same manner as for subclassing.

**Definition of Delegation** We say that a state machine  $\mathcal{D}$  is derived by *delegating* to two (or more) machines  $\mathcal{B}$  and  $\mathcal{C}$  if there is an *embedding* (i.e., a mapping) from the sets  $Trans_b$  and  $Trans_c$  to the set  $Trans_d$ , such that both maps from  $Trans_b \rightarrow Trans_d$  and  $Trans_c \rightarrow Trans_d$  individually satisfy the conditions for subclassing. However, at any particular state of the program, only one of the possible embeddings of the components may be true.

**Definition of Genericity** We say that a machine  $\mathcal{D}$  is an instantiation of state machine template  $\mathcal{B}$  if there is a one-one mapping between the states, events, actions, and transitions of  $\mathcal{B}$  and  $\mathcal{D}$ .  $\mathcal{B}$  is called a template because its actions are parametrized by states (and possibly other objects). Otherwise, it is just like a state machine.

Now that we have defined our constructions as maps between transitions, we investigate what sorts of maps can be implemented efficiently.

---

<sup>4</sup>We assume that the notation used to describe state machines enforces unambiguous names for events and states as necessary.

## 5 Efficient Implementation

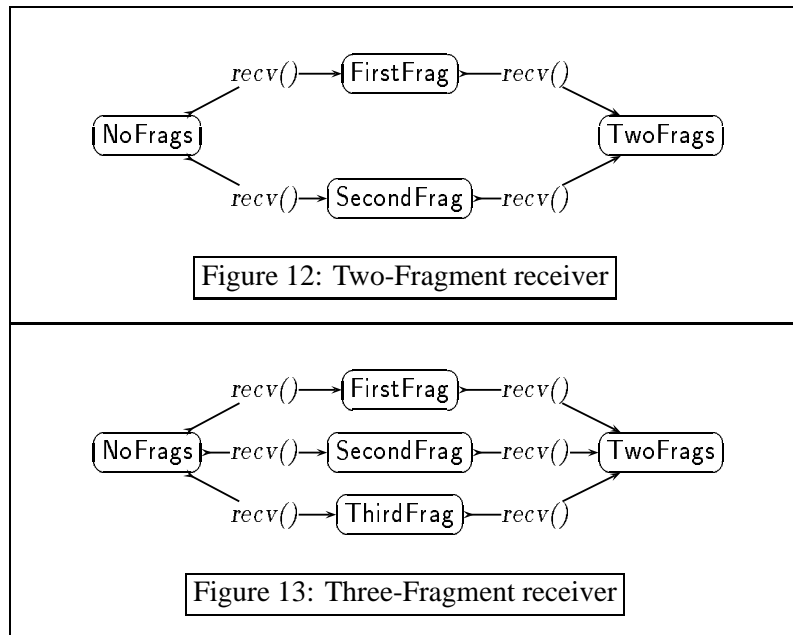
We show how embeddings can be programmed so that event dispatch requires only one addition and a single table lookup. We first show that the StateMap scheme of Figure 5 cannot be used to program all embeddings. We also discuss limitations due to non-determinism.

**Need for StateMap Parameters** Suppose that two transitions  $\langle S, evA(), T \rangle$  and  $\langle S, evB(), T \rangle$  of a base machine  $\mathcal{B}$  are embedded into two derived state machine  $\mathcal{D}$  transitions  $\langle S_d, evA(), T_d \rangle$  and  $\langle S_d, evB(), U_d \rangle$  respectively. Then, the destination state in the derived machine depends on the *event* invoked. So we need a *different StateMap* in  $S_d$  depending on the *event*. Therefore, unlike Figure 5, events of a single state may require multiple StateMaps that are *parameters* to action methods. Now, only non-deterministic transitions pose a problem.

**Nondeterminism** State transitions are non-deterministic if two transitions with the same source state and event have different destination states. Figure 12 shows a networking machine that can receive messages transmitted as two fragments. The non-determinacy is due to `NoFragments::recv()`, since the criterion used to choose between the destination states is hidden. Now suppose the Two-Fragment machine in Figure 12 is subclassed to create the Three-Fragment machine in Figure 13. In Figure 13, `NoFragments::recv()` has more possible destinations — we say that the derived machine Three-Fragment *increases non-determinacy*. A StateMap cannot be used to *increase* non-determinacy, since one entry of an array can map to only one destination state; therefore, method `NoFragments::recv()` has to be redefined for the Three-Fragment machine, although it may invoke the base-class methods to achieve some reuse. In the following, we assume that derivations do not increase non-determinacy.

### 5.1 Implementation

In this section, we show that all embeddings apart from those that increase non-determinacy can be efficiently implemented. We statically *precompute* the states to be returned, methods to be dispatched, etc. Then, instead of programming individual StateMaps



in C++ (Section 2) and using C++’s implicit VTables, we construct our own composite VTables and State-Maps for all states together, considered as a single object. Then we have:

**THEOREM 1 (EmbeddingsUsingArrays)** *Given an embedding from a base to a derived machine, method calls on derived states may be dispatched with one addition operation, followed by one table lookup, regardless of the height of the subclassing, composition, or delegation hierarchy.*

We use induction on the hierarchy of embeddings. In our base case, we indicate how to achieve two step dispatch in a state machine without any ancestors. In the inductive step, we indicate how the tables of ancestors can be extended in a derivation so that no additional lookup is required.

## 5.2 Base Case: Simple Machine without Ancestors

We show how to optimize method calls and state lookup for a single machine that has no ancestors.

### 5.2.1 Optimizing Method Dispatch

Method calls on states may originate from the object that uses the state machine (*Outer* calls), as well as

from other states (*Inner* calls).

**Optimizing Outer Calls** Using the implementation of Figure 4, an *Outer* call to *pageAccess()* such as

```

Page::pageAccess()
{ state = state->pageAccess(this); }
  
```

involves two dereferences, one for the *state* variable and another to look up *pageAccess* in the VTable of *state*. To optimize, we concatenate the VTables for all state objects of a machine in a composite *CVTable*. The *current state* is represent just by an integer. Now, methods are dispatched simply by indexing into the *CVTable* using the current state and the method index, which involves only one addition and one lookup.

**Optimizing Inner calls** Dispatching an *Inner* call to *pageAccess()* such as

```

Unmapped::pageAccess(Page *p)
{ p->getPhysicalPage(); p->readDisk();
  return map->Mapped()->pageAccess(p); }
  
```

in Figure 4 involves one lookup in a *StateMap* followed by a VTable lookup. To optimize, we maintain separate inner call vtables (called *IVTables*) that have precomputed pointers to methods actually invoked in *Inner* calls. Thus we need only one table lookup. The

particular IVTable depends on the event dispatched (i.e., method invoked).

### 5.2.2 Optimizing State Lookup

The StateMap to be used for a transition is determined during the Outer call, along with the IVTable. Thus, the next state in an embedding can be returned with only one table lookup. This completes the description of the *base case*.

### 5.3 The Inductive Step: Machines With Ancestors

For the inductive step, consider two machines  $\mathcal{B}$  and  $\mathcal{D}$ , such that  $\mathcal{D}$  is derived from  $\mathcal{B}$ .  $\mathcal{B}$  itself is presumably derived from other machines; we assume that events of  $\mathcal{B}$  are dispatched as for the base case. We show how to construct tables for  $\mathcal{D}$ , such that every table  $T_b$  in  $\mathcal{B}$  has a corresponding table  $T_d$  that *extends*  $T_b$ . Thus, all indexing operations from the source code for  $\mathcal{B}$  work without change, and have no extra overhead.

**Subclassing** As an example, let  $\mathcal{B}$  be the VM machine from Figure 1, and  $\mathcal{D}$  be the COW machine from Figure 2. The following arguments hold for these two machines. We have three cases:

**CVTABLE** Since every state of machine  $\mathcal{D}$  responds to every event of  $\mathcal{B}$ , VTables for state objects of  $\mathcal{D}$  extend the VTables for state objects of  $\mathcal{B}$ . Hence, the CVTables of  $\mathcal{D}$  are an extension of the CVTable of  $\mathcal{B}$ , in that for every state-event combination of  $\mathcal{B}$ , there is one for  $\mathcal{D}$ . Furthermore, there is no additional overhead to lookup the CVTable of  $\mathcal{D}$ .

**STATEMAP** Suppose that a transition  $t_b$  of  $\mathcal{B}$  is mapped to some transitions  $u_d$  and  $v_d$  of  $\mathcal{D}$ . As long as the subclassing does not increase non-determinacy,  $u_d$  and  $v_d$  have different source states. So we create two copies of the StateMap used for  $t_b$  for each derived source state, and replace the destination as required. Thus, the tables are extensions with no extra lookup overhead.

**IVTABLE** IVTables are like StateMaps, except that they point to methods. Thus, the IVTables can be extended just like StateMaps.

Thus, we have shown that state machines can be inherited with fast dispatch.

**Composition** Composition is defined (Section 4) as a set of embeddings each of which defines a subclass, so we apply the above technique for each component and the composite.

**Delegation** Delegation is also defined as a set of embeddings each defining a subclass. So for each delegatee, we construct tables as above, and switch tables if the delegatee changes.

**Genericity** All instantiations of a template can share the structure of the template. Since instantiations only differ in the parameters, we simply use method “pointers” from the CVTable and IVTable as indices into *Instantiation Tables*. Every instantiation uses a distinct instantiation table. Instantiation tables can also be eliminated by making copies, so, any particular generic machine can be sped up if necessary (see Section 3.3).

### 5.4 Space Overhead

In the standard non-object-oriented table based implementation of deterministic state machines, we need one table that has the next state for every combination of current state and event, and one for pointers to methods that define the actions.

Our technique requires a minimum of one-and-half times the space of standard table method for the CVTable, plus space for StateMaps and IVTables. In the worst case, every combination of source state and event needs its own StateMap. Additionally, each StateMap might contain entries for every single state in the base machine. Similarly, in the worst case, every method may invoke all methods of all states. However, usually StateMaps and IVTables can be shared, and do not refer to every other state and method, so we need little space. For example, in Figure 5, we need only two StateMaps with four entries each, although the derived machine has four states, each with three new methods. Also, some space is saved due to code reuse.

## 5.5 Limitations

As we have seen, the StateMap technique cannot completely eliminate method redefinition for non-deterministic transitions. A more pragmatic limitation is that for our C++ like [7] *implementation*, the structure of state machines must be statically declared. Of course, a more dynamic Smalltalk like implementation is straightforward to build.

## 6 Miscellaneous topics: subtyping, nested states, and generators

This section summarizes some issues that are not discussed in detail due to space restrictions. More details may be found in [22].

**Subtyping and Substitutability** The definitions from Section 4.1 enforce structural similarities, but we consider behavioral similarities separately using the Liskov-Wing behavioral subtyping [15] approach (LWA). Briefly, the idea behind LWA is to specify state invariants, method pre-and-post conditions and history properties of state transitions, and show that derived machines respect the properties of the base machine. For example, the VM machine (Figure 1) states have the invariant that “a virtual page is either mapped to only one physical page, or not mapped at all”. Clearly, the COW machine (Figure 2) states respect this invariant. Similarly, we can specify the other properties to prove that a COW machine IS-A VM machine. This technique can be extended for composition to show that the DVM machine IS-A VM machine.

**Nested States** The implementation technique from Section 5 can be extended to program nested states as in StateCharts [10]. Essentially, superstates with nested states can be treated as a collection of states. Thus, each state in the collection is represented using tables as in Section 5, and the tables are switched at run-time to represent nested state transitions.

**Automated Generation** We have programmed a simple yacc-like generator that takes state machine

descriptions with support for subclassing, composition, etc. C++ code for all of the virtual memory state machines described in the paper is generated automatically. Compared to the original non-object-oriented implementation of VM and DVM, the equivalent new implementation requires only half as many lines of generated C++ code. For new protocols, reuse is even greater, so that our latest implementation supports several new protocols but is still smaller than the original. The input to the generator is only 40% of the generated code, since it consists mostly of the method code and eschews C++ declarations and other repetitive code. We are also developing a graphical version of the generator.

## 7 Related Work

The reification technique (RT) has been invented many times [16, 6, 14, 13], usually for the purpose of “changing the class of an object”. Behavior abstractions [14] and Enabled sets [28] introduce *become* and *replace* constructions that hint at our general concept of embeddings. Johnson and Zweig [13] and Liu [16] have applied RT program network protocols. Liu also suggests the use of a CVTable. The relation between states and classes is explored in [20]. A discussion similar to that in Section 2 is found in [18]. Taivalsaari [27] suggests that objects have modes, and transitions between modes are governed by a transition function. Chambers’ predicate classes [4] are similar to modes or states but more dynamic. Our implementation technique can be extended to predicate classes using methods that automatically select the next state. In particular, our definitions of subclassing, composition, etc., can be applied with predicate classes, and easily admit more dynamic implementations. An early version of subclassable state machines was described in [23].

Our object-oriented constructions extend Harel’s StateCharts [10]. Other object-oriented extensions include *ObjectCharts* [5], *ROOMCharts* [25], *DisCo* [12], *ObjCharts* [9], etc. In ObjectCharts, ROOMCharts and ObjCharts, state machines are inherited by adding new states and actions, or by refining existing actions. Substitutability is achieved essentially by

requiring that the subtype traces should extend the behavior of the parent. In DisCo, inheritance is defined as importing class definitions.

In contrast, our approach parallels that used in standard object-orientation — we define a notion of Self that (1) is a guide for modifications and reasoning, (2) makes constructions such as delegation natural, and (3) leads to a highly efficient implementation technique. Further, we use the simple Liskov-Wing behavioral subtyping (Section 6) as opposed to traces. However, unlike StateCharts, etc., at present we do not consider concurrency.

## 8 Conclusion

We have described what appears to be the first general definition of inheritance in state machines; its distinguishing features are our notion of Self for state machines, and the concept of Embeddings. With our technique, state machine designers may easily construct complex state machines using object-oriented techniques, while sharing code for actions. Our efficient implementation technique permits the use of complex hierarchies in practice. Event dispatch with our approach is only slightly more expensive than a C++ virtual function dispatch.

The techniques presented in this paper open further avenues of investigation. For instance, we have found simple solutions to the so-called inheritance anomaly [17] based on state machine inheritance. Our notion of *Self* seems to lead naturally to recursively typed extensions of Nierstrasz's regular types [19] and inheritance in path expressions [3]. State machines are used in specifying *interfaces* [29], so subclassing or composition of interfaces might be investigated using our definitions. It may be possible to apply our notion of Self to frameworks, where objects are interrelated by relationships other than BECOMES-A. Besides our four constructions, it may be interesting to consider higher-order state machines that manipulate other machines. Finally, work remains to be done in adapting state machine design tools for object-oriented designs. Expressing embeddings graphically might be an interesting challenge.

*Acknowledgments* Prof. Ralph Johnson taught us RT

and critiqued several versions of this paper. Prof. Michael Loui, Ellard Roush, Amitabh Dave, Mohlalefi Sefika, Tin Qian, David Putzolu, Mark Monnin, Brian Foote and Don Roberts helped improve the presentation. Hermann Hüni and Antoni Bieri commented on an earlier version of this paper. We also thank the anonymous referees for helpful comments.

## References

- [1] M. J. Bach. *The Design of the UNIX operating system*. Prentice Hall, Englewood Cliffs, New Jersey, 1986.
- [2] Grady Booch. *Object-oriented Analysis and Design with Applications*. Benjamin/Cummings, California, 1994.
- [3] R.H. Campbell. The Specification of process synchronization by Path-Expressions. In *Lecture Notes in Computer Science*, pages 89–102, 1974.
- [4] Craig Chambers. Predicate classes. In *Proceedings of the European Conference on Object-Oriented Programming*, July 1993.
- [5] Derek Coleman, Fiona Hayes, and Stephen Bear. Introducing Objectcharts or how to use Statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.
- [6] Stephen R. Davis. C++ objects that change their types. *The Journal of Object-Oriented Programming*, pages 27–32, July/August 1992.
- [7] Margaret Ellis and Bjarne Stroustrup. *The C++ Annotated Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.
- [8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.
- [9] Dipayan Gangopadhyay and Subrata Mitra. ObjChart: Tangible specification of reactive object behavior. In *Proceedings of the Euro-*

- pean Conference on Object-Oriented Programming, number 707 in Lecture Notes in Computer Science, pages 432–457. Springer-Verlag, New York, 1993.
- [10] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, August 1987.
- [11] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.
- [12] Hannu-Matti Järvinen and Reino Kurki-Suonio. DisCo specification language: Marriage of actions and objects. In *International Conference on Distributed Computing Systems*, pages 142–157, 1991.
- [13] Ralph E. Johnson and Jonathan M. Zweig. Delegation in C++. *The Journal of Object-Oriented Programming*, pages 31–34, Nov/Dec 1991.
- [14] Dennis G. Kafura and Keung Hae Lee. Inheritance in actor based concurrent object oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'89)*, pages 131–145. Cambridge University Press, 1989.
- [15] Barbara Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (to appear)*, 1994. Available from <http://www.cs.cmu.edu:8001/afs/cs/user/wing/www/home.html>.
- [16] Chung-Syan Liu. An object-based approach to protocol software implementation. In *Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 307–316, 1994.
- [17] Satoshi Matsuoka, Kenjiro Taura, and Akinori Yonezawa. Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, volume 28 of *SIGPLAN Notices*, pages 109–126. ACM Press, October 1993.
- [18] John D. McGregor and Douglas M. Dyer. A note on inheritance and state machines. *ACM Sigsoft Notes*, 1994.
- [19] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, volume 28 of *SIGPLAN Notices*, pages 1–15. ACM Press, October 1993.
- [20] Alexander S. Ran. Modelling states as classes. In *Proceedings of the Technology of Object-Oriented Languages and Systems Conference*, 1994.
- [21] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenzen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
- [22] Aamod Sane. Object-Oriented State Machines: Subclassing, Composition, Delegation, and Genericity. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1994. See <http://choices.cs.uiuc.edu/sane/home.html>.
- [23] Aamod Sane and Roy Campbell. Subclassing state machines. Technical Report UIUCDCS-R-92-1773, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.
- [24] Aamod Sane, Ken MacGregor, and Roy Campbell. Distributed virtual memory consistency protocols: Design and performance. In *Second IEEE workshop on Experimental Distributed Systems*, 1990.
- [25] Bran Selic. An efficient object-oriented variation of the Statecharts formalism for distributed real-time systems. In *CHDL'93 IFIP Conference on Hardware Description Languages and Their applications*, pages 26–28, April 1993.
- [26] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles: Modeling the World in States*. Yourdon Press Computing Series. Yourdon Press (Prentice Hall, Englewood Cliffs, New Jersey), 1992.

- [27] Antero Taivalsaari. Object-oriented programming with modes. *The Journal of Object-Oriented Programming*, pages 25–32, June 1993.
- [28] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled-sets. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, SIGPLAN Notices, pages 103–112. Association for Computing Machinery, ACM Press, October 1989.
- [29] Daniel M. Yellin and Robert E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications 94*, pages 176–190, 1994.