An Extensible State Machine Pattern for Interactive Applications

Brian Chin and Todd Millstein

Computer Science Department
University of California, Los Angeles
{naerbnic, todd}@cs.ucla.edu

Abstract. The *state design pattern* is the standard object-oriented programming idiom for implementing the state machine logic of interactive applications. While this pattern provides a number of advantages, it does not easily support the creation of extended state machines in subclasses. We describe the *extensible state design pattern*, which augments the traditional state pattern with a few additional constraints that allow subclasses to easily add both new states and new events. Further, we observe that *delimited continuations*, a well-known construct from functional programming languages, supports *state refinement* in subclasses as well as the modular expression of control flow in the presence of interaction. We illustrate our pattern in the context of Java, leveraging its generics to obviate the need for dynamic typecasts and employing a small library that implements delimited continuations. We have used our pattern to simplify and modularize a widely used application written by others.

1 Introduction

Interactive applications are those that repeatedly accept input from and produce output to an external entity. Typically the output produced depends upon the current input along with the current state of the application, and this output may in turn affect the next input. Many common application domains are fundamentally interactive, including servers, user interface programs, and computer games. In addition, more traditional applications often include an interactive component. For example, a program whose behavior is configured by an XML file might interactively parse the file through the event-driven Simple API for XML (SAX) [17].

The logic of an interactive application essentially takes the form of a state machine, and the standard way to implement this logic in an object-oriented (OO) language is with the *state design pattern* [9]. This pattern reifies each state as a distinct class, which has one method for each possible external event. The state machine class maintains a field containing the current state, and all external events are forwarded to the current state to be handled appropriately. Handling an event may result in some output and additionally update the current state.

The state design pattern has a number of advantages. Because the current state is represented explicitly as an object, there is no need to manually test the current state of the machine when an event occurs. Instead, the state object is sent an ordinary message send upon an event, and each state "knows" how to appropriately respond to each kind of event. Further, each state class can naturally encapsulate its own data (i.e., fields), which is less error prone than storing all necessary data in the state machine itself.

While the state design pattern simplifies the creation of a *new* state machine, even simple ways in which one might want to extend an existing state machine in a subclass are difficult to implement without code duplication and/or unsafe features like type casts. State logic is inherently difficult to reuse via standard mechanisms like method overriding, since the logic of the machine is fragmented across multiple cooperating event handlers. The state design pattern exacerbates this problem by fragmenting the application logic across several interdependent classes. As a result, the traditional benefits of object-oriented software reuse mechanisms are not readily applicable to interactive applications.

In this paper, we present an extension of the state design pattern that we call the *extensible state design pattern* (Section 2). In addition to the requirements of the basic state pattern, we impose new rules on how state machines and state classes should be structured. Obeying the rules allows subclasses to modularly and safely extend the original state logic in a variety of desirable ways. This pattern is implemented within vanilla Java 1.5, however, the pattern is not Java-specific and could be implemented in other OO languages. The pattern relies on the *generics* found in both Java and C#; an implementation in C++ is possible using templates but would have weaker type-correctness guarantees.

Using our pattern, subclasses of a state machine can easily add new states to the machine and override existing states to have new behaviors (Section 2.1), as well as add new kinds of events that the extended state machine can accept (Section 2.2). These tasks are similar to those in the *expression problem* identified originally by Reynolds [16] and named by Wadler [20]. Torgersen [19] provides several solutions to the expression problem in Java, which make heavy use of generics. Our solution borrows ideas from his "data-centered" solution but is specialized for the domain of the state design pattern, which allows for a simpler solution without loss of functionality. For example, since states are not a recursive datatype, we do not require the sophistication of F-bounded polymorphism [3].

The state pattern additionally has several extensibility requirements that have no analogue in the expression problem. For example, we would like to allow a subclass to easily "interrupt" the existing state logic, insert some additional logic, and later resume the original state logic. This natural idiom can be seen as the interactive equivalent of a subroutine call. It can also be used to express a form of hierarchical state machines, whereby a state of the superclass is implemented in the subclass as its own state machine. Further, traditional control flow logic such as subroutines and loops are difficult to express modularly even within a single state machine, due to the need to pass control back to the environment. For instance, if a state machine must wait for an event in the middle of a loop,

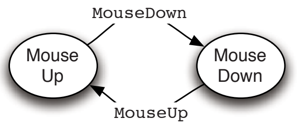

Fig. 1. The Base State Machine

that loop must be unrolled and split between multiple classes, obfuscating the original intent and introducing new possibilities for error.

We observe that *delimited continuations* [7], a well-studied language feature from the functional programming community, naturally supports modular expression of traditional control flow in the presence of interaction. We have implemented a form of delimited continuations as a small Java library with a simple API (Section 4), and we incorporate the usage of this API as constraints in our extensible state design pattern. We illustrate how this API and the associated constraints overcome all of the difficulties described above and provide several other benefits (Sections 2.3 and 3).

To validate our design pattern, we have used it to refactor a widely used application written by others (Section 5). This application, JDOM [12], is an XML parser that creates a DOM tree by using a SAX model parser. JDOM was originally implemented as a monolithic class that used several fields to encode properties of its current state. We refactored its implementation to employ our design pattern, which greatly simplified the logic and made it significantly more readable. Further, we demonstrate the extensibility benefits of our pattern by structuring the refactored code as two state machines: a class that supports basic XML parsing and a subclass that supports more advanced features of XML and has the same functionality as the original JDOM implementation.

2 The Extensible State Pattern

In this section we build up our extensible state machine pattern in stages, beginning with the standard state design pattern [9]. As a running example we consider a state machine for a simple user interface, along with several desired extensions to this state machine. Each stage in our discussion will refine the design pattern to obey new constraints necessary to enable a particular kind of extensibility. Our example sometimes sacrifices realism for simplicity, but it represents the kinds of tasks which are needed in UIs in general.

In our first user interface, there is a window containing a single button. The state machine logic should simply cause a function `triggerButton` to be invoked whenever the button is clicked. The `InputState` interface at the top of Listing 1 shows the three events that can occur based on a user's actions. Clicking a button actually consists of two events, a mouse down followed by a mouse up, both of which need to occur inside the bounds of the button. The diagram for this state machine is depicted in Figure 1.

```
interface InputState {
    void MouseUp(Point at);
    void MouseDown(Point at);
    void MouseMotion(Point from, Point to);
}

class InputStateMachine {
  // standard currState members
  private InputState currState = new MouseUpState();
  public InputState getCurrState() {
    return currState;
  }
  protected void setCurrState(InputState newState) {
    currState = newState;
  }

  // state class definitions
  protected class MouseUpState implements InputState {
    public void MouseDown(Point at) {
      if (buttonShape.contains(at)) {
        setCurrState(new MouseDownState());
      }
    }

    public void MouseUp(Point at) {}
    public void MouseMotion(...) {}
  }

  protected class MouseDownState implements InputState {
    public void MouseUp(Point at) {
      if (buttonShape.contains(at)) {
        setCurrState(new MouseUpState());
        triggerButton();
      }
    }

    public void MouseDown(Point at) {}
    public void MouseMotion(...) {}
  }

  // forwarding methods and other members...
}
```

Listing 1. The base code for the UI example

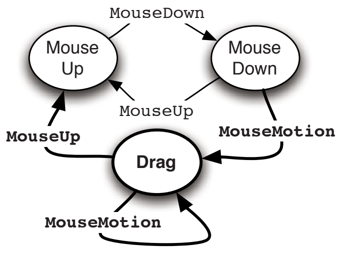

Fig. 2. Adding the Drag State

The rest of the code in Listing 1 uses the standard state design pattern to implement the desired functionality. The `InputStateMachine` class maintains a field `currState` representing the current state of the machine. There is one state class per state in our machine. The `MouseUpState` represents the situation when the mouse is currently up, and similarly for `MouseDownState`. We define these classes as inner classes to allow them access to the state machine's members. Forwarding methods (not shown) pass signaled events to `currState`, which does the main work of the state machine.

In the rest of this section, we illustrate how to sequentially extend our example in three stages:

1. We will add basic drag-and-drop capabilities, allowing the user to click-drag the button in order to move it around. Releasing the mouse after a drag will not trigger the button.
2. We will add an event to handle keyboard presses, which can change the button's color. The user may modify the button's color while dragging it.
3. We will add a feature to hit a designated button during a drag, which will bring up a dialog box with information about the dragged object. When the dialog box is dismissed, the drag will continue.

2.1 Adding and Overriding States

As the diagram in Figure 2 shows, implementing drag-and-drop functionality requires the creation of a new state, to represent the situation when we are in the middle of a drag. The state machine should move to this state upon a `MouseMotion` event when the mouse is down on the button, and subsequent `MouseMotion` events should be used to move the dragged button.

The state design pattern makes adding new states straightforward: a subclass `DragStateMachine` of `InputStateMachine` can simply contain a new inner class `DragState` to represent the `DragState`. `DragStateMachine` can similarly contain a subclass `DragMouseDownState` of `MouseDownState`, which overrides the implementation of `MouseMotion` to move to the dragging state as appropriate.

Unfortunately, these changes alone will not affect the state machine logic, since the state machine is still creating instances of `MouseDownState` rather than `DragMouseDownState`. We can of course solve this problem by code duplication,

```
class InputStateMachine {
  // standard currState members
  private InputState currState = makeMouseUpState();
  // ...

  // factory methods
  protected InputState makeMouseUpState() {
    return new MouseUpState();
  }

  protected InputState makeMouseDownState() {
    return new MouseDownState();
  }

  // state class definitions
  protected class MouseUpState implements InputState {
    public void MouseDown(Point at) {
      if(buttonShape.contains(at)) {
        setCurrState(makeMouseDownState());
      }
    }
    public void MouseUp(Point at) {}
    public void MouseMotion(Point from, Point to) {}
  }

  // ...
}
```

Listing 2. The base state machine with factory methods added

for example by creating a subclass `DragMouseUpState` of `MouseUpState`, which reimplements the `MouseDown` method to instantiate `DragMouseDownState`. However, this approach is tedious, error prone, and non-modular. This problem leads to the first new constraint for our design pattern:

> *Constraint*: There must exist a consistent way of creating states that will allow future extensions to override the implementation of a state class.

To satisfy the constraint, we introduce *factory methods* [9] in the base state machine, as shown in Listing 2. The state machine logic must never directly instantiate state classes, but instead always go through the factory methods. For example, the `MouseUpState`'s `MouseDown` method now invokes `makeMouseDownState` to create the new state.

Given this extension to the state design pattern, implementing drag-and-drop functionality is straightforward, as shown in Listing 3. We define a new class `DragState` as well as a subclass `DragMouseDownState` of `MouseDownState`. To incorporate `DragMouseDownState` into the state machine logic, we simply override the corresponding factory method for that state. We also create a new

```
class DragStateMachine extends InputStateMachine {
  // overridden factory methods
  protected InputState makeMouseDownState() {
    return new DragMouseDownState();
  }

  // new factory methods
  protected InputState makeDragState() {
    return new DragState();
  }

  // subclassed state classes
  protected class DragMouseDownState extends MouseDownState {
    public void MouseMotion(Point from, Point to) {
      setCurrState(makeDragState());
    }
  }

  // new state classes
  protected class DragState implements InputState {
    public void MouseUp(Point at) {
      setCurrState(makeMouseUpState());
    }

    public void MouseMotion(Point from, Point to) {
      buttonShape.move(from, to);
    }

    public void MouseDown(Point at) {}
  }
}
```

Listing 3. The drag-and-drop extension

factory method for the dragging state, so that `DragStateMachine` itself satisfies our constraint. In this way, the new state machine can itself be seamlessly extended by future subclasses. We will maintain this *hierarchical* nature of the design pattern throughout.

To summarize, we add the following rules to the standard state design pattern, in order to support new states:

– Each state class should have an associated factory method in the state machine class.
– State objects must always be instantiated through their factory method.

2.2 Adding Events

As the diagram in Figure 3 shows, in order to implement our second extension we need to respond to a new kind of event, representing a keyboard press. It is

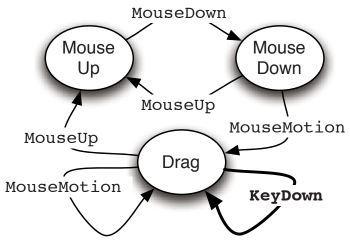

Fig. 3. Adding the KeyDown Event

natural to incorporate this event through an extension to the `InputState` event interface:

```
public interface KeyState extends InputState {
  public void KeyDown(Key key);
}
```

Now a subclass `KeyStateMachine` of `DragStateMachine` can subclass each state class to add a `KeyDown` method and implement this new interface. However, all of the factory methods are declared to return an `InputState`, as is the `currState` field. Therefore, the `KeyStateMachine` will have to use type-unsafe casts from `InputState` to `KeyState` whenever it needs to make use of the new `KeyDown` method. If the implementer forgets to subclass one of the state classes appropriately, this error will only manifest as a runtime `ClassCastException`.

The underlying problem is that the state interface is set in stone in the base state machine. To be able to update the state interface without typecasts, our pattern should obey the following constraint:

Constraint: Each state machine must abstract over the events it responds to. While it may require that certain events exist, it may not limit what events can be added by future extensions.

Generics provide a natural way to satisfy this constraint. Rather than hard-coding the interface for events as `InputState`, we use a type variable to represent the eventual interface to be used, as shown in Listing 4. The `State` type variable replaces all previous occurrences of `InputState`. The `State` type variable is declared to extend `InputState`, so the implementation of the state machine can assume that at least the three events in `InputState` will be handled.

Since the factory methods no longer know which concrete class will actually meet the abstract interface `State`, they can no longer have a concrete implementation and are instead declared `abstract` (making the entire class abstract as well). As a result, the `InputStateMachine` class can no longer be instantiated directly. Rather, we must *concretize* the state machine, as shown in Listing 5; this new class is identical in behavior to our original version of the UI from Listing 1. Concretization serves two purposes. First, it fixes the set of events by instantiating the `State` type variable with an interface. Second, it fills in all of

```
abstract class InputStateMachine<State extends InputState> {
  // standard currState members
  private State currState = makeMouseUpState();
  public State getCurrState() {
    return currState;
  }
  protected void setCurrState(State newState) {
    currState = newState;
  }

  // factory methods
  protected abstract State makeMouseUpState();
  protected abstract State makeMouseDownState();

  // ...
}
```

Listing 4. `InputStateMachine` modified for adding events

the factory methods by instantiating classes that meet this interface. Because `ConcreteInputStateMachine` explicitly defines the state interface, it effectively terminates future extensions being made from it. Of course this does not prevent further extensions derived off of `InputStateMachine`.

The entire logic of the state machine is still contained within the abstract state machine class. For example, the class in Listing 4 will contain the definitions of the `MouseUpState` and `MouseDownState` classes that we have seen earlier. The uniform usage of factory methods and the `State` type variable allow the definitions of these state classes to remain unchanged. For example, a call of the following form is the idiomatic way to change states and requires no typecasts within the context of the class in Listing 4:

```
setCurrState(makeMouseUpState())
```

Extending the state machine is now accomplished by subclassing from the abstract state machine class. Listing 6 contains an updated version of our drag-and-drop state machine. The body of this class is identical to that of Listing 3, except that the `State` variable is used in place of `InputState` and the factories are abstract. Keeping this class abstract allows it to be uniformly extended, as we will do next. Naturally, the concretized drag-and-drop state machine would instantiate the `State` variable as `InputState` and add the necessary implementations of the factory methods.

Finally, Listing 7 shows how to use our pattern to easily add new events. The `State` variable is given the new bound `KeyState`, which indicates that the state machine must handle the `KeyDown` event in addition to the others. Accordingly, the existing state classes are subclassed in order to provide appropriate `KeyDown` implementations. The concretized version of this state machine (not shown) will instantiate `State` with `KeyState` and override all of the factory methods to

```
class ConcreteInputStateMachine extends
InputStateMachine<InputState> {
  protected InputState makeMouseUpState() {
    return new MouseUpState();
  }

  protected InputState makeMouseDownState() {
    return new MouseDownState();
  }
}
```

Listing 5. The concretized `InputStateMachine`

```
abstract class DragStateMachine<State extends InputState>
  extends InputStateMachine<State>
{
  // new factory methods
  protected abstract State makeDragState();
  // ...
}
```

Listing 6. The `DragStateMachine` extension modified for adding events

```
abstract class KeyStateMachine<State extends KeyState>
  extends DragStateMachine<State>
{
  public class KeyDragState extends DragState implements KeyState {
    public void KeyDown(Key key) {
      if (key.equals(COLOR_KEY))
        changeButtonColor(key);
    }
  }

  // default implementation
  public class KeyMouseUpState
    extends MouseUpState implements KeyState
      { public void KeyDown(Key key) {} }
  // same for others ...
}
```

Listing 7. Adding a new event in a state machine extension

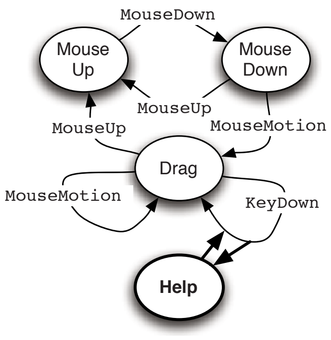

Fig. 4. Interrupting the Drag

instantiate the new state classes. Unlike with the original pattern, no type casts are necessary, and the Java typechecker will signal an error if one of the state classes is not properly handling the new event.

To summarize, we add the following rules to our design pattern, in order to support new events:

– A state machine must define a type variable that is bound by the currently known state interface.
– This type variable must be used uniformly in place of any particular state interface.
– All factory methods are declared abstract.
– A state machine must be concretized before it can be used, by fixing the state interface type and implementing the factory methods.

2.3 Adding "Subroutines"

With the above modifications to our pattern, we can modularly add both new states and new events. While these abilities allow essentially arbitrary modifications to the base state machine, there is a common extensibility idiom that deserves special support. It is often useful to "interrupt" an existing state machine at some point, insert some new state logic, and later "resume" the original state machine where it left off. Intuitively, this is the interactive equivalent of a subroutine call, and it also naturally represents a form of hierarchical state refinement, in which a state of the superclass is implemented as its own state machine in the subclass.

A case in point is our final extension, shown pictorially in Figure 4. While dragging an object, a user can press a specified key to bring up a dialog box about the entity being dragged. Another key press will dismiss the dialog box, at which point the drag should be resumed. Effectively, the drag state is being hierarchically refined. We could implement this extension using the above techniques, but manually interrupting and resuming the drag is tedious. Further, that approach requires care to ensure that the state of the drag upon resumption is identical to the state before the interruption. For example, in general it may not be sufficient to simply create a brand new instance of `DragState` with which to resume the drag, since that could discard important state from the original drag. This brings us to our final constraint:

> *Constraint*: Each state transition should be able to be interrupted and later resumed by a subclass.

As mentioned above, the interruption is akin to a subroutine call in traditional program logic. We might therefore attempt to satisfy our constraint by allowing the base state machine to include a call to a dummy method `interruptKeyDown` within each `KeyDown` method:

```
public void KeyDown(Key key) {
  // ...
  interruptKeyDown(key)
  // ...
}
```

The location for this call is decided in the superclass. Now, we can override `interruptKeyDown` in subclasses in order to perform the interruption. Unfortunately, such an interruption would be forced to complete entirely within the current state transition, before control is returned to the event sender. Therefore, such an approach does not allow interruptions that require further user interaction, as is required in our example.

One way around this problem is to capture the part of the `KeyDown` method after the interrupt as an explicit function that can be called at will by subclasses. Java's `Runnable` interface provides a solution:

```
public void KeyDown(Key key) {
  // ...
  interruptKeyDown(key, new Runnable() {
    public void run() {
      // ... rest of the transition after the interrupt
    }
  });
}

public void interruptKeyDown(Key key, Runnable next) {
  next.run();
}
```

By default, `interruptKeyDown` simply invokes the given `Runnable` immediately, thereby executing the rest of the transition. However, a subclass can override the method to properly perform the interruption:

```
public void interruptKeyDown(Key key, Runnable next) {
  if (key.equals(HELP_KEY)) {
    setCurrState(makeHelpState(next));
  } else {
    super.interruptKeyDown(key, next);
  }
}
```

In the above code, if the help key is pressed, then we move to the new help state (not shown). That state is passed the given `Runnable`, so it can properly resume the original transition when the dialog box is dismissed by the user. If a key other than the help key is pressed, then a `super` call is used to perform the original transition as usual. With this approach, a state machine designer can easily declare points in each state transition that are interruptible, allowing future extenders to insert arbitrary state logic without breaking the original state machine's invariants.

There are two problems that need to be addressed in this approach. First, the above code still requires the subclass to explicitly set the state back to the drag state upon a resumption of the original transition. To address this problem, we require each event handler to always end by setting its state appropriately, *even if the state does not change*. With this rule, we can be sure that the original code will set its state appropriately upon being resumed. To satisfy our rule, the original code for `KeyDown` will be modified as follows:

```
public void KeyDown(Key key) {
  // ...
  interruptKeyDown(key, new Runnable() {
    public void run() {
      // ... rest of the transition after the interrupt
      setCurrState(this);
    }
  });
}
```

The call to `setCurrState` ensures that we always return to the original drag state after the dialog box subroutine completes.

Second, the use of simple functions (i.e., `Runnables`) to capture the code after the interruption has a number of limitations. Since a runnable can only capture the code within a single method, it has to be created in the top-level event handler method, rather than in some auxiliary method. Similarly, these interrupt points cannot easily occur within control structures like loops or conditionals, since the resulting runnable would be stuck in a particular scope and therefore unable to capture the entire rest of the computation. What we need is a uniform way to save the entire state of the computation after an arbitrary interrupt point.

We discovered that *delimited continuations* [7,2,8], a language feature developed in the functional programming community, does exactly this. Programmers can declare a *reset point* at any point in the code, which has no semantic effect. However, if a *shift* is later executed, then the entire execution stack up to the

```
public void KeyDown(Key key) {
  reset {
    //...
    shift (continuation) {
      interruptKeyDown(key, continuation);
    }
    //...
    setCurrState(this);
  }
}

public void interruptKeyDown(Key key, Continuation cont) {
  cont.execute();
}
```

Listing 8. Example use of delimited continuations

most recent reset is popped off and saved as a continuation. A block of code provided with the shift is subsequently executed and is passed the continuation, which can be invoked to restore the original computation.

For example, the shift-reset version of our `KeyDown` method is shown in Listing 8. It has the same semantics as the earlier code, but it avoids the limitations mentioned above. The shift can occur anywhere in our code, even in methods called by `KeyDown` or inside of control structures. Further, the "rest" of the computation can be nicely kept outside of the shift block, unlike with runnables.

We have created a simple Java library that implements delimited continuations, which is discussed in Section 4. The library allows the code to be written essentially as shown above, except that `reset` and `shift` are method calls into the library. For ease of presentation, we continue to use the prettier syntax.

Listing 9 shows how to use delimited continuations to implement our final state-machine extension. The relevant portion of the `KeyStateMachine` has been modified to satisfy the new constraint. The `KeyDown` method properly ends by setting the state. The `getThis` factory method is necessary in order to satisfy the typing constraints introduced by abstracting on the `State` type variable; the concretization of this class will implement `getThis` appropriately. The `KeyDown` method uses a shift to support interruption by subclasses. As mentioned earlier, the state machine forwards each event to `currState`. Therefore, it is natural to put a *reset* in each such forwarding method, as shown at the bottom of the figure, thereby alleviating the need for resets within the state classes.

Listing 10 shows our final state machine extension. We override `interrupt-KeyDown` in the dragging state in order to move to the new help state, rather than simply calling the continuation. The new state stores the continuation and opens up the dialog box. When any key is pressed subsequently, the dialog box is closed and the continuation is invoked, in order to resume the drag.

To summarize, we add the following rules to our state design pattern, in order to support state-logic interruptions:

```
abstract class KeyStateMachine<State extends KeyState>
  extends DragStateMachine<State>
{
  public abstract class KeyDragState extends DragState implements KeyState {
    public abstract State getThis();

    public void KeyDown(Key key) {
      if (key.equals(COLOR_KEY)) {
        changeButtonColor(key);
      }
      shift (continuation) {
        interruptKeyDown(key, continuation);
      }
      setCurrState(this.getThis());
    }

    protected interruptKeyDown(Key key, Continuation cont) {
      cont.execute();
    }
  }

  public void KeyDown(Point at) {
    reset {
      this .getCurrState().KeyDown(at);
    }
  }
}
```

Listing 9. The Key state machine with inserted interrupt-point

– The last command on each path through an event handler must either be a `setCurrState` call or an invocation of a continuation.
– Each forwarding method in a state machine class should set a reset before forwarding an event to the current state.
– An *interrupt point* consists of a shift placed anywhere inside code that is part of an event handler. The associated code block contains a call to an interrupt method, to which it passes the created continuation as well as any auxiliary information.
– The default behavior for an interrupt method is to immediately call the continuation which it is passed.

3 Interrupt Points Explored

This section discusses how our novel notion of interrupt points may be used in our pattern to gain even more flexibility, giving several examples to illustrate their expressiveness in a variety of dimensions.

```
abstract class HelpStateMachine<State extends InputState>
  extends KeyStateMachine<State>
{
  // new factory methods
  public abstract State makeHelpState(Continuation cont);

  public abstract class HelpDragState extends KeyDragState {
    public void interruptKeyDown(Key key, Continuation cont) {
      if(key.equals(HELP_KEY)) {
        setCurrState(makeHelpState(cont));
      } else {
         super.interruptKeyDown(key, cont);
      }
    }
  }

  // new state class
  public abstract class HelpState implements KeyState {
    private Continuation cont;

    public HelpState(Continuation cont) {
      showHelpWindow();
      this.cont = cont;
    }

    public void KeyDown(Key key) {
      closeHelpWindow();
      cont.execute();
    }
    // other events with the default body ...
  }
}
```

Listing 10. Our extension using the added interrupt-point

3.1 Returning Values from Interrupt Points

So far shifts have been used only as control structures, copying the stack into a continuation to return in the future. Our library also allows a shift to return a value. The following code illustrates a simple example:

```
String name = shift(Continuation<String> k) {
    k.execute(“Hello World!”);
}
```

As usual, the shift saves the current execution state in the continuation `k` and executes its body. The type of the continuation indicates that it expects a `String` as an argument. Accordingly, the continuation is invoked with a string literal in the shift block. This argument becomes the value of the entire `shift` expression, so the above code causes `name` to have the value `"Hello World!"`.

The ability for “interrupters” to easily pass values back to the interrupted state logic is often extremely useful. Such values can be used to change the behavior of the original state logic or to allow that logic to declaratively gather necessary data from its extensions. Our case study in the next section uses this feature of shifts to good effect.

3.2 A Stack of Interrupted States

Since any state that stores a continuation from an interrupt point may itself be interrupted, it is easy to form an arbitrarily long chain of states, each of which has been interrupted by the next state on the chain. In essence, this is the interactive equivalent of a run-time call stack. Executing a shift that transitions to a new state and passes the current continuation to that state has the effect of pushing that new state onto the call stack. Invoking a continuation has the effect of popping the top state off the call stack. This ability makes the state machine powerful enough to declaratively encode a pushdown system. Our case study in the next section relies on this technique to handle parsing of arbitrarily nested XML data.

Similar functionality could be implemented by having each state keep a reference to the previous state, given to it at creation time, forming a reference stack that does not use delimited continuations. When a machine wants to transition back to a previous state, it just calls `setCurrState()` with the stored state pointer. In the pure state machine case, where the only purpose of state transitions is to end up in the specified state, this would work fine. In real-world cases, when state transitions can have general Turing-complete code on them, delimited continuations allow clean-up code to be run after the interrupt point is returned to, such as that which may be desired in a locking protocol. Further, the clean-up code could even be used to decide which state should come next, based on the current context.

3.3 After-the-fact Interrupt Points

In our example in the previous section, the implementer of the base state machine anticipated the need for an interrupt point in the `KeyDown` event handler. However, subclasses can easily add new interrupt points after the fact, for use both within that subclass and within any future extensions. Since our pattern requires that the base state machine wrap each event handler call with a reset, any shifts within the dynamic extent of an event handler are always well defined. For example, if `KeyDown` did not contain a shift, a subclass could simply override `KeyDown` and add one. We make use of this ability in our case study in the next section.

3.4 Interrupt Points and Information Hiding

In the traditional state design pattern the current state object must maintain all of the data associated with the current execution state. If any data is needed

```
public void KeyDown(Key key) {
  DelimitedContinuation.Reset(new ResetHandler() {
    public void doReset() {
      //...
      DelimitedContinuation.Shift(new ShiftHandler<Unit>() {
        public void doShift(Continuation<Unit>() cont) {
          interruptKeyDown(key, cont);
        }
      });
      //...
      setCurrState(this);
    }
  });
}

public void interruptKeyDown(Key key, Continuation<Unit> cont) {
  cont.execute(null);
}
```

Listing 11. Version of Listing 8 using our API

in future states, it must be explicitly passed along to a new state whenever a state transition occurs. Thus states may have to store data that they don't need in order to pass it on to states that may use it later. Aside from being tedious, this also results in a loss of modularity, since data has to be available where it logically should never be manipulated.

Interrupt points provide a convenient solution to this problem. A continuation uniformly stores all current data (indeed, all data on the stack up to the recent reset) and encapsulates it as a single value. Therefore, a state need only accept a continuation in order to maintain all of the data potentially needed in the future, and the state only needs to explicitly maintain the data that it actually manipulates. When the continuation is eventually invoked, the data in the continuation is restored and made available to the state logic that has been resumed.

4 Implementation

As previously mentioned, we implemented delimited continuations as a Java library. Each continuation is implemented as a thread, which is a simple way to save the current execution state. A continuation thread `waits` on itself until it is invoked. At that point the continuation thread is `notify`ed so it can run, and the calling thread in turn `waits` on the continuation thread. When the continuation thread is to return, the reverse logic happens. In this way we ensure a deterministic handoff of control between threads.

Our library has a simple API. Listing 11 shows how Listing 8 looks using the API. `Reset` is a static method on the `DelimitedContinuation` class. It takes a

ResetHandler as an argument, whose doReset method provides the implementation of the reset block. Shift is handled analogously. The ShiftHandler is parameterized by the type of the result, as discussed in Section 3.1. The Unit type admits only the value null, thereby acting similar to void. The doShift method is provided the continuation thread as an argument. When the continuation is eventually invoked, the Shift method returns the value the continuation was passed, and the code proceeds as usual.

Our library approach to implementing delimited continuations has a few limitations. First, a continuation cannot be invoked more than once, and doing so results in a dynamic error. Second, resets prevent exceptions from continuing up the stack, thereby violating normal exception semantics. Others have considered direct support for continuations in the Java virtual machine [5], which could resolve these limitations.

5 Experience

JDOM [12] is a Java implementation of the Document Object Model (DOM) for XML, which represents XML data as a tree of objects. Clients can then use this tree to easily access the XML data from within Java programs. JDOM's implementation parses XML files using a SAX parser, which reads an XML file and reports events to an instance of JDOM's SAXHandler class, such as the start of a new element, one by one. The SAXHandler object incrementally builds the DOM tree in response to each event from the parser. As such, SAXHandler is a real-world example of an interactive software component.

The original SAXHandler implementation is written as a single monolithic class, rather than using the state design pattern. We refactored the code to use our extensible state design pattern, creating explicit state classes. To illustrate the extensibility provided by our pattern, we implemented the functionality of SAXHandler in two stages. First we implemented a base state machine that can build the DOM tree for basic XML documents. Then we created a subclass of this state machine to handle more advanced features of XML, including entities, Document Type Definitions (DTDs), and CDATA blocks. This class has the same functionality as the original SAXHandler class.

5.1 Base State Machine

The original SAXHandler class implements four interfaces, which contain the various parsing events that must be handled. Our basic refactored version of SAXHandler implements only the ContentHandler interface, which provides events for, among other things, the beginning and end of the XML document, the beginning and end of an XML element, and character data within an element.

This state machine (depicted in Figure 5) is fairly simple. There are three main states: The first is the initial state. On a startDocument event, it enters the main parsing state. When the document is done, it gets sent the endDocument event which causes it to enter the Document Complete state. There is one more

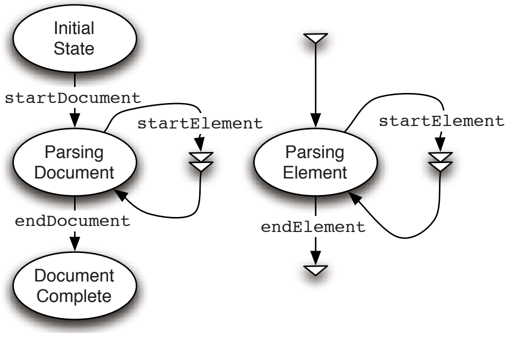

Fig. 5. The State Machine for the Simple SAX Handler

state devoted to parsing XML elements, which we will describe in more detail shortly.

Implementing this state machine in our pattern was straightforward. The most interesting part is the need to handle arbitrarily nested elements. Effectively, the state machine needs to maintain a stack of elements that are currently in the process of being parsed. In the original code, this stack was maintained explicitly, and integer fields were used to keep track of the current nesting depth during parsing.

Our use of interrupt points provides a much more natural solution. We employ our aforementioned "subroutine" idiom to parse a single element. This pattern is indicated in Figure 5. The interrupt point (represented by the double triangle) is the entrance to the subroutine that begins at the small triangle at the top, entering the "Parsing Element" state. When this state receives an `endElement` event, it will exit the subroutine environment and return the constructed element, allowing the remainder of the "calling" code to complete (in this case, adding the returned element to the document). The parsing element state will also enter into the same subroutine upon receiving the `startElement` event, causing a recursive call. This recursion is what gives rise to the implicit stack-like nature of this idiom.

Listings 12 and 13 show the code that implements this approach. When a `startElement` event occurs, we invoke the `readElement` method shown in Listing 12. This method shifts the event handler's execution, stores it into a continuation `k`, and transitions into a `ParsingElementState` object, which stores the continuation (in field `prevCont`) for later use. Recall that our pattern places a reset at the beginning of each event handler, so this shift is well defined. The `ParsingElementState` builds up the current element (in field `currElement`) as it receives `characters` events. If it receives a `startElement` event, then it invokes `readElement` to recursively interrupt execution in order to parse the nested element. Finally, as shown in Listing 13, when the `ParsingElementState` receives the `endElement` event it invokes the stored continuation in order to resume execution of the interrupted state machine, passing the parsed element back. This value becomes the return value of the shift from `readElement`.

```
public Element readElement(String name) {
    return shift (Continuation<Element> k) {
        setCurrState(makeParsingElementState(name, k));
    }
}
```

Listing 12. The `readElement()` method

```
public void endElement(String name) {
    prevCont.execute(currElement);
}
```

Listing 13. The `endElement()` event handler for the `ParsingElementState`

In addition to methods representing possible events, the `ContentHandler` interface contains a method `getDocument`. This method should return the root of the DOM tree if parsing has completed and `null` otherwise. This method does not update any local state and hence is not part of the state logic of the machine. Therefore, it is safe to implement it as a regular method, which does not conform to the rules of our design pattern. For instance, it does not begin with a reset nor end by updating the state. Our design pattern naturally accommodates such methods, which query the state of the machine but do not update it.

5.2 Extended State Machine

Our subclass of the above state machine class adds support for the events in the `DeclHandler`, `DTDHandler`, and `LexicalHandler` interfaces. These interfaces respectively add support for XML entities, DTDs, and CDATA blocks. With the addition of support for these events, our version of `SAXHandler` implements all of the functionality of the original class. While for brevity's sake we implemented these aspects in a single extension, we could just as easily have created one extension for each of these aspects.

In total, we added four new states and support for 12 new events. We also used four interruption points to insert "subroutines" in the original logic. Our extensible state design pattern made these additions straightforward. The most inconvenient part was the addition of the new events, which required subclassing each of the existing state classes in order to add the new methods. If Java had multiple inheritance, we could create a class `DefaultState` which contains default handlers for the new events, and each new state class could then inherit from both the appropriate old state class as well as `DefaultState`. Because Java lacks multiple inheritance, each new state class instead has its own implementation of each of the new events, thereby incurring some code duplication.

```
//...
if (atRoot) {
  document.setRootElement(element);
  atRoot = false;
} else {
  factory.addContent(getCurrentElement(), element);
}
currentElement = element;
```

Listing 14. A snippet of `startElement()` from the original `SAXHandler` implementation

We briefly discuss each of the three new pieces of functionality in turn. XML entities are names that can be given to a block of XML data. When the name is later referenced, it has the effect of inserting the associated data at the current point, similar to a `#include` directive in C. Accordingly, when the SAX parser encounters a reference to an entity, it sends events that correspond to the entity's associated data.

The original implementation of `SAXHandler` allowed the client code to decide whether to handle entities properly or to simply ignore them. This was accomplished via a boolean field `suppress`, which was consulted within each event handler to determine whether to handle the current event or not. Our implementation uses a more declarative approach. When we receive a `startEntity` event in the `ParsingElementState`, we check the `suppress` field once. If the client has configured us to expand all entities, we simply continue as usual. Otherwise, we transition to a new `SuppressedState`, which simply ignores all events.

When the `SuppressedState` receives an `endEntity` event, we must transition back to the state we were in before the most recent `startEntity` event. Effectively, the logic for suppressing entities interrupts the ordinary flow of the state machine and later resumes it. Therefore, an interrupt point is the natural approach for implementing this extension. Accordingly, the `ParsingElementState`'s `startEntity` method uses a `shift` to transition to the `SuppressedState`:

```
shift (Continuation<Unit> cont) {
    setCurrState(makeSuppressedState(cont));
}
setCurrState(this.getThis());
```

Upon an `endEntity` event, the `SuppressedState` invokes the given continuation in order to resume the original state logic. After invoking the continuation, the last statement above is executed, in order to return the state machine to the proper state before returning control to the SAX parser.

Both DTDs (inline declarations of the XML schema) and CDATA blocks (inline escaped text) were parsed in a similar manner. A new state was defined for each, which was able to accept the events necessary to parse their respective structure. The parsing of a CDATA block produces a value, so we implemented a `readCDATA` method in the same mold of the `readElement` method shown in Listing 12.

5.3 Comparison

It is instructive to compare our refactored version of `SAXHandler` with the original one. The original class maintained its state through many fields, including seven boolean variables and an explicit stack for keeping track of the incomplete elements. The event handlers were typically rife with `if` statements dispatching on the aforementioned boolean fields to implement state-like behavior. For instance, Listing 14 shows a snippet from the `startElement` event handler which used the `atRoot` field to decide which implementation to use. Thus implementation for two states was put into the same method, making it hard to understand. In contrast, our pattern allowed us to separate out code associated with different states, with each state class maintaining its own fields. For example, our version of Listing 14 has each branch as an event handler in a distinct state.

Our code was longer than the original code. Some of this was due to boilerplate code that, to a practiced eye, could be quickly understood. Some of it was due to the extra classes and methods which our pattern requires. The base class in our version has 388 non-comment non-whitespace lines and the extension has 600, while the original JDOM code has 424 non-comment non-whitespace lines. Excluding boilerplate (forwarding methods, empty event handlers, and factory declarations), our numbers are 270 for the base and 330 for the extension.

We believe that the improved readability and extensibility of the reimplemented code outweighs the increase in code length. The mental overhead of the pattern could be reduced by using a static checking framework such as JavaCOP [1] to automatically ensure that the pattern's constraints are obeyed. It could also be possible to automatically generate much of the boilerplate code, given a high-level description of the state machine.

6 Related Work

The *expression problem* [16,20] highlights the difficulty of adding both new operations and new classes to an inheritance hierarchy in a statically typesafe manner, and many solutions have been proposed. Our work borrows from solutions proposed in Java [19] and in Scala [14], both of which use the idea of concretizing a generic abstract class with a trivial concrete subclass that instantiates the type parameters. Our pattern additionally introduces interrupt points via delimited continuations as a form of extensibility and modular control flow in the face of interactive logic.

Family polymorphism [6] is an inheritance scheme that allows a group of classes to be extended simultaneously, enabling each of the extensions to explicitly use the new features of the other extended classes. This allows for much more powerful interrelationships between the classes in the group as compared to our state class extensions. Several languages such as gbeta [10] and Scala [18] implement a version of it. Family polymorphism could be used to make our pattern more lightweight. For instance, some forms of family polymorphism can obviate the need for factory methods by making constructors virtual. Even so, our pattern remains simple and can be implemented in vanilla Java 1.5.

Delimited continuations [7] are a language feature derived from classic continuations that limit the amount of remaining execution they save and can be called without losing the current state of execution. A great deal of work has been done in the functional community detailing properties and implementation issues of delimited continuations [2,8] . To our knowledge, the use of delimited continuations to achieve a common form of extensibility for state machines has not previously been investigated.

The PLT Scheme web server [13] uses continuations to store the state of HTTP sessions. This allows them to maintain state while transferring information over the otherwise stateless HTTP. This approach is similar to our implicit stack approach. We additionally identify the synergy between delimited continuations and inheritance in OO languages, in order to support natural forms of state machine extensibility, and we codify this idiom in a general design pattern.

Others have recently added direct support for various forms of continuation in the Open Virtual Machine [15] for Java [5]. By leveraging their work, we may be able to avoid the overhead of switching thread contexts in our implementation, thus improving our performance and making our delimited continuation library more powerful.

We previously described *ResponderJ* [4], an extension to Java that allows state logic to be implemented as ordinary control flow interspersed with coroutine-like event-dispatch blocks called `eventloop`s. *Actors* [11] implement a similar mechanism using the closures and pattern matching in Scala. Both features are primarily targeted at improving the state logic of a single state machine rather than at easily creating new state machines from old ones. However, careful planning and leverage of ordinary method overriding can be used to achieve the forms of extensibility that our pattern supports. For example, encapsulating each `eventloop` in ResponderJ in its own method allows subclasses to override this method in order to achieve the effect of replacing an existing state. ResponderJ is a fairly large language extension, while our new pattern is implementable in vanilla Java.

7 Conclusion

We have defined the extensible state design pattern, which adds a small number of requirements onto the traditional state design pattern. By requiring a state machine to obey extra constraints, we make it possible for subclasses to easily and flexibly extend the state machine in several dimensions. Our pattern is implementable in Java, and we have also shown how a library based on the notion of delimited continuations can give the pattern more power. Our experience indicates that our pattern's new requirements are easy to respect and that the pattern provides commonly desired forms of extensibility in a practical manner.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0427202 and CCF-0545850, as well as by a generous gift from Microsoft Research. Thanks to Robby Findler and the anonymous reviewers for useful feedback on this work.

References

1. Andreae, C., Noble, J., Markstrum, S., Millstein, T.: A framework for implementing pluggable type systems. ACM SIGPLAN Notices 41(12), 57–74 (2006)
2. Biernacki, D., Danvy, O., Shan, C.: On the static and dynamic extents of delimited continuations. Sci. Comput. Program 60(3), 274–297 (2006)
3. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: Proc. of 4th Int. Conf. on Functional Programming and Computer Architecture, FPCA 1989, London, September 11-13, 1989, pp. 273–280. ACM Press, New York (1989)
4. Chin, B., Millstein, T.D.: Responders: Language support for interactive applications. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 255–278. Springer, Heidelberg (2006)
5. Dragos, I., Cunei, A., Vitek, J.: Continuations in the java virtual machine. In: Proceedings of the Second Workshop on Implementation, Compilcation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS 2007) (2007)
6. Ernst, E.: Family polymorphism. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 303–326. Springer, Heidelberg (2001)
7. Felleisen, M.: The theory and practice of first-class prompts. In: POPL, pp. 180–190 (1988)
8. Flatt, M., Yu, G., Findler, R.B., Felleisen, M.: Adding delimited and composable control to a production programming environment. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2007) (2007)
9. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Massachusetts (1995)
10. GBeta home page, `http://www.daimi.au.dk/~eernst/gbeta`
11. Haller, P., Odersky, M.: Event-based programming without inversion of control. In: Lightfoot, D.E., Szyperski, C.A. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 4–22. Springer, Heidelberg (2006)
12. JDOM home page, `http://www.jdom.org`
13. Krishnamurthi, S., Hopkins, P.W., McCarthy, J., Graunke, P.T., Pettyjohn, G., Felleisen, M.: Impelementation and Use of the PLT Scheme Web Server. In: Higher-Order and Symbolic Computation (2007)
14. Odersky, M., Zenger, M.: Independently extensible solutions to the expression problem. In: Proc. FOOL 12 (January 2005)
15. Ovm home page, `http://www.ovmj.org`
16. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to type abstraction. In: Schuman, S.A. (ed.) New Directions in Algorithmic Languages, pp. 157–168. IRIA, Rocquencourt (1975)

17. The Simple API for XML (SAX) home page, `http://sax.sourceforge.net`
18. The Scala language home page, `http://scala.epfl.ch`
19. Torgersen, M.: The expression problem revisited. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 123–146. Springer, Heidelberg (2004)
20. Wadler, P.: The expression problem. Email to the Java Genericity mailing list (December 1998)