Document #	433UD022	Activity Paper	
Title	A Design Pattern for Implementing Robust Protocol Machines in Java		
Author / Body	L. Fritsch (SRB), M. Schunter (UDO)		
Editor	M. Schunter (UDO)		
Reviewer			
Date	28 Aug 1998		
Status	Draft	Vers. 2	SEMPER internal
CEC / TA id	none		

Activity Paper

433UD022
28 Aug 1998

Version 1

- *Submission version.*

Version 2

- *Added a section on the efficiency of the pattern.*

Table of Contents

1 Introduction _______ 3

2 The Pattern "Extended Finite State Machine" _______ 4

2.1 Intent _______ 4

2.2 Motivation _______ 4

2.3 Operation _______ 6

2.4 Properties _______ 6

3 Java Representation of the Protocol Machine _______ 7

3.1 The Extended Finite State Machine Pattern _______ 7

3.2 Messages _______ 7

3.3 States _______ 8

3.4 Message Processing, State Transitions, Default Error Handling _______ 9

3.5 Design Options _______ 11

3.5.1 Extending the Machine _______ 11

3.5.2 Time-Outs _______ 11

3.5.3 Multi-Party Protocols _______ 11

3.6 Sample Code _______ 11

3.7 Efficiency _______ 13

4 Acknowledgments _______ 13

5 Bibliography _______ 14

A Design Pattern for Implementing Robust Protocol Machines in Java

Lothar Fritsch
Universität des Saarlandes
Im Stadtwald 45
66123 Saarbrücken
<fritsch@fsinfo.cs.uni-sb.de>

Matthias Schunter
Universität Dortmund
Informatik 6
44221 Dortmund
<schunter@acm.org>

Abstract
Specifying and implementing distributed protocols in Java imposes many problems on programmers: A clear definition of the messages to be handled and dealing with unexpected messages are fundamental requirements for any robust implementation.
In this article, we sketch the well-known formalization of distributed protocols as extended finite state machines and show how to implement them in Java in a robust and extensible way. Exploiting object oriented design, processing of protocol events is done by a convenient automated dispatcher in contrast to traditional error-prone event loops. Furthermore, we describe several options of the pattern that increase the robustness of the implementation in different environments.

1 Introduction

A distributed protocol is a collection of machines that cooperate by sending and receiving messages. Each machine takes messages as input, performs some local computations that may change the machine's state, and finally outputs messages. Examples for such protocols are communication protocols [HoGr_89], payment protocols [AJSW_97], cryptographic protocols [Schn_96], or other commerce protocols [ScWa_97]. Note that each protocol may have many steps of progress. The protocols described in [PfSW_98], for example, each have up to eight states and the protocol has about fifteen possible messages which have to be handled by the machines.
A major problem when designing such protocols is to specify exactly which messages exist, which messages are expected in a given state and finally what happens if an unexpected message occurs in any given state. Even if the specification is complete, in practice, most implementations of security-critical protocols have failures [Neum_95]. A common problem in the implementation results from incomplete handling of all possible and unforeseen messages in each state. As a result, the protocol is neither robust, nor does it follow its specification.
In order to simplify implementing correct and robust distributed protocols, we describe a pattern for distributed protocol machines in Java which has been used for payment and communication protocols in SEMPER [ScWa_97]. The main goal is to encourage a clear definition of all states and messages as well as the behavior when processing any message in any state. Our design reflects this goal by defining the known messages as well as default error-handling in a superclass of all states. Each

subclass then inherits the automated error handling while defining the intended behavior for expected messages. This way, a complete specification in one place guarantees robustness in the protocol implementation.

2 The Pattern "Extended Finite State Machine"

2.1 Intent

Our goal is to provide a pattern for extended finite state machines which can be combined to distributed protocols. Requirements are: automated enforcement of a well-defined reaction to all messages as well as ease of use.

2.2 Motivation

An extended finite state machine [Lync_96] is a finite state machine[1] where each message and each state may contain a finite number of variables and the state transitions not only change to another state out of the finite set but also perform arbitrary computations on the variables contained in the state, the message received, and the message sent (the "extension"). The purpose of modeling the infinite state of a Turing machine as a combination of a small finite set of states together with an assignment of values to a given finite set of variables is to enable more intuitive protocol specifications: Each of the finite number of states and messages has an intuitive meaning whereas the variables make the model as powerful as Turing machines.

For a distributed two-party protocol, two such machines are then connected by sending all messages output by one machine into the other machine and vice versa. This communication between the participants is not considered part of the extended finite state machine since we aim at a protocol which is independent from the means of communication.

[1] A finite state machine is a tuple (S, M, s_0, F, d) where S is a finite set of states, M is a finite set of inputs (i.e., messages), s_0 is the starting state, F are the final states, and *d:SxM SxM* is the transition function which get a state and a message as input and produces a follow-up state and a message to be sent as output.

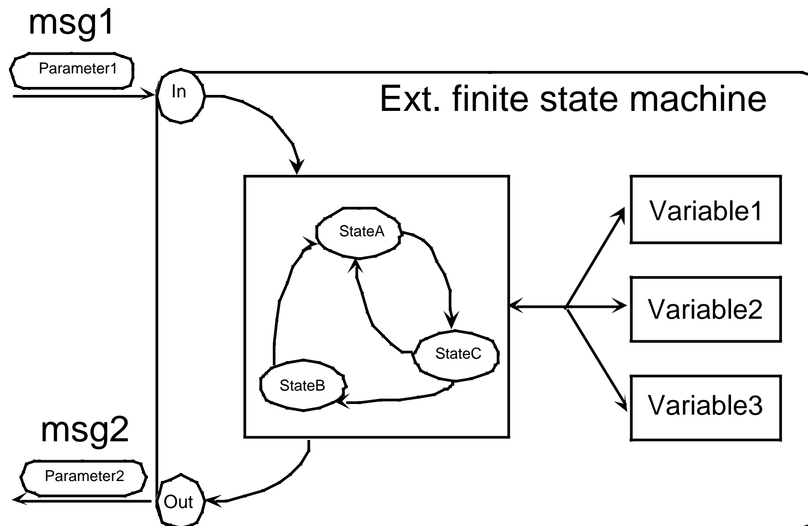

Figure 1: An Extended Finite State Machine

The implementation of a distributed protocol with communicating machines in Java requires an implementation of the machine and of the communication. While the possibilities of implementing communication are numerous, we want to focus on ways to enforce the protocol and minimize the chances for undetected protocol failures or other pitfalls for programmers and users of the protocol. Therefore, our requirements for a robust distributed protocol are:

- well-defined states,
- well-defined messages, and
- a well-defined reaction on any kind of message in any state.

As an example that guides the reader through the Java pattern, we use a manual car engine gear box with two speeds, neutral and reverse gear. Shifting from reverse to the driving speeds and vice versa requires passing the neutral state, otherwise the old-fashioned gearbox will break (i.e., enters the *Broken* state). The gear shifting rules constitute the protocol. We model the gear box as an extended finite state machine, and the shifting is implemented as the messages the extended finite state machine receives. Two equivalent descriptions of the contained finite state machine are presented in Figure 2 and Figure 3[2].

[2] Note that depicting machines as tables only works for the underlying finite state machine but does not allow to describe machine or message variable dependent state transitions. This is not sufficient for all machines since, for example, the resulting state may depend on the contents of a received message or the current machine variable assignment. More powerful specification techniques for extended finite state machine are, e.g., described in [Hogr_89].

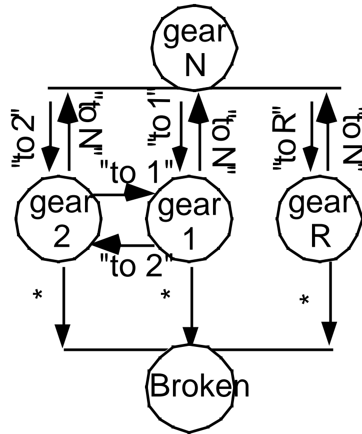

Figure 2: State transition diagram of a finite-state machine for a gear machinery (Text in Quotes are message labels, "*" denotes all unspecified messages).

Message Current state	"to R"	"to N"	"to 1"	"to 2"
R	*R*	*N*	*BROKEN*	*BROKEN*
N	*R*	*N*	*1*	*2*
1	*BROKEN*	*N*	*1*	*2*
2	*BROKEN*	*N*	*1*	*2*
BROKEN	*BROKEN*	*BROKEN*	*BROKEN*	*BROKEN*

Figure 3: Alternative description of the gear machinery as a transition table, i.e., resulting states for each state and message.

2.3 Operation

First, instantiate a machine. Then, input received or instantiated messages (possibly containing data) to it, and analyze the message returned as a response. In order to produce the output message, the state object internal to the machine will process the message, do computations on the state variables, instantiate a message to be returned, and change the state.

2.4 Properties

The behaviour of the machine is well-defined by the extended finite state machine's states (that include the transition function. Message processing and error handling also happen in the state classes.

Protocol failures[3] are caught and can possibly be recovered if states of the extended finite state machine reflect recoverable protocol transactions and recovery has been properly specified.

3 Protocol failures are the receiption of messages not expected in the current state and the receiption of machine-internal error messages (defined in the machine) .

The protocol can be modified or extended by in a variety of ways which are described in Sections 3.5.

3 Java Representation of the Protocol Machine

We now describe the Java representation of the extended finite state machine. In general, the class *Machine* contains the data of the machine on which the protocol computations take place, the subclasses of *Message* define all messages and their data content, the class *State* defines the default error handling, and its subclasses correspond to the states and define the behavior of the machine, i.e., the code executed for each message expected in each particular state of the machine.

3.1 The Extended Finite State Machine Pattern

The classes of the extended finite state machine and their relations are depicted in Figure 4. Each extended finite state machine has a state, as well as the data common to all states on which the machine performs computations while processing messages.
Intuitively, the machine has some code for handling each specific message in each specific state (i.e., each cell in the table in Figure 3). We structured these pieces of code after the current state of the machine (the rows of the table), i.e., each subclass of *State* defines how to process the messages expected in this state. Unexpected as well as unknown messages[4] are handled by the *State* superclass.
We now describe the message and state classes and then how message processing and exception handling work.

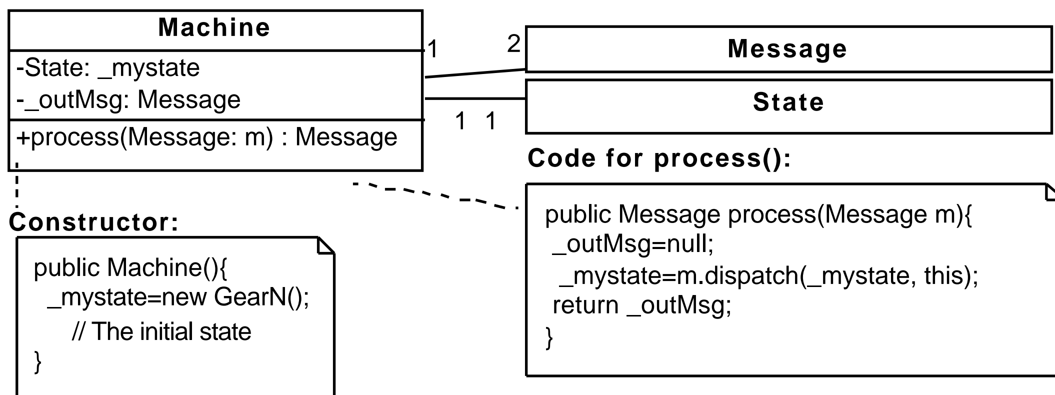

Figure 4: The machine and its classes in UML notation [FoSc_97].

3.2 Messages

Figure 5 shows an abstract message superclass and the message subclasses of the car gear example. The *Message* superclass defines protocol parameters such as message

4 "Unknown" messages are messages that have not been declared for processing in the *State* superclass.

sequence numbers or the addressee of messages in multi-party protocols. An inherited subclass of *Message* may define message-specific parameters.
Furthermore, the Message superclass defines an abstract method *dispatch()* for dispatching itself to the appropriate method of the current state. *dispatch()* is called by the machine after receiving a message.

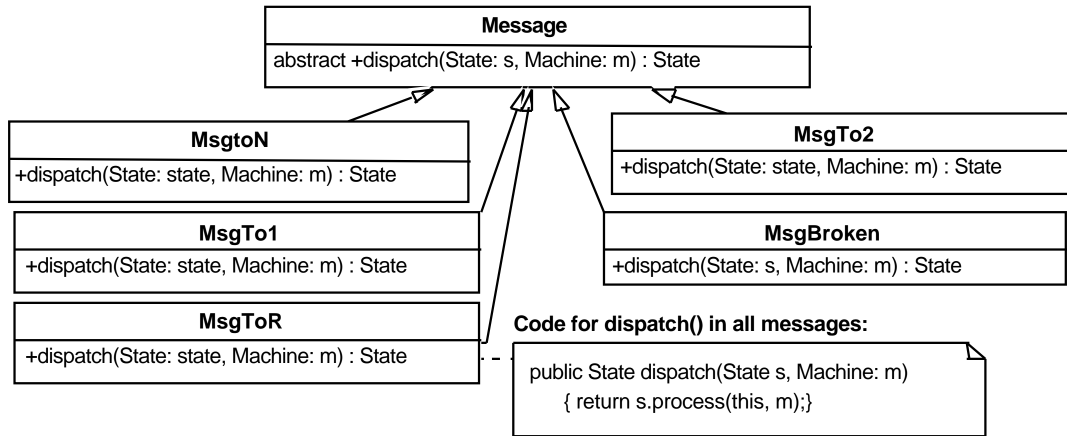

Figure 5:The Message class hierarchy.

Note that this method is overloaded with identical code in all subclasses. This has proven essential in Java because the language lacks run-time subclass awareness when subclasses are addressed under their superclass type. Without the subclass-local dispatcher, no automatic selection of the appropriate processing methods in the states is possible: A message read from a network is de-serialized as *Object* and then is cast to *Message* without knowing the subclass. Casting the message to its correct subclass is left to the message object's *dispatch()* method (which knows about its subclass type). That ensures our concept of subclassed messages where the environment of the machine need not be aware of the type of the message. See [GJS_96] for the Java method invocation procedure.
In our example, each *MsgToX* tells the gearset to shift into Gear *X*. When shifting to neutral, one may tell the machine to switch the engine off by setting the flag contained in *MsgToN.* States processing *MsgToN* can turn the engine off if they want. Switching into any other gear then switches the engine on again.

3.3 States

The *State* class and its subclasses for the car gear example depicted in Figure 6 define the actual behavior of the machine. This class hierarchy contains all protocol states as well as the complete protocol behavior as it defines the automaton's transition function in its message processing methods.

Note that Java does not allow to override methods in subclasses (i.e., introduce new messages): Instead, when selecting the method whose signature fits the call, it appears that Java first searches for the most specific method in the *State* superclass and then checks whether this particular method has been overloaded without searching for further subclass methods that have a more precise or more general signature for the

object to be passed into the state. As a workaround, all messages to be handled in any subclass have to be defined in the superclass of the states.

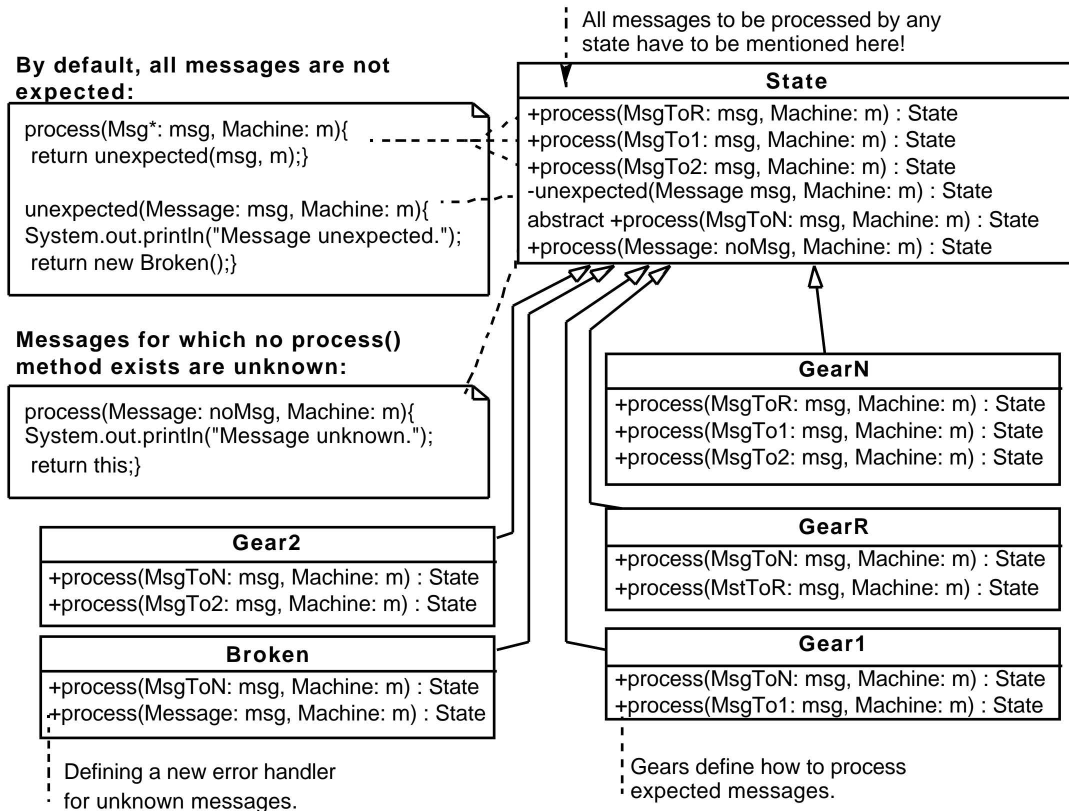

Figure 6: The State class hierarchy.

3.4 Message Processing, State Transitions, Default Error Handling

When the process method of the machine is called with a received message, the machine asks the message to dispatch itself to the current state for processing. This is done in the *dispatch()* method of the message. The indirection is needed since the type of the message *msg* sent into the machine is *Message*. So calling *process(msg)* will always invoke the *process(Message msg, Machine m)* method instead of the specific method for this subclass of message. If, on the other hand, *process()* is called by the message itself, the correct method of the current state is selected. While processing the message, the *process()* method can change the variables of the machine and return the subsequent state. Finally, the machine changes to the new state and outputs the message generated by the *process()* method.

The superclass *State* common to all states defines how messages are handled by default. Given a subclass of *State*, a message can be handled in the following ways:

- *Known and Expected messages:* For these messages, the state defines a *process()* method handling the messages. For accessing the machine's variables, it receives a reference to the machine as additional input. The method returns the subsequent state. If a message is required to be expected and processed

individually by each state, its *process()* method must be declared *abstract* in the superclass. Depending on the intended scope of a process method, there are several ways to declare them:

- *Processing a particular message in a particular state (a table cell in Figure 3):* A method *process(MsgType mess, Machine m): Message* for the particular message type is declared in the particular state (such individual processing in each state can be enforced by declaring the method abstract in the state superclass).
- *Identical processing of a particular message in all states (a column in Figure 3):* A method *process(MsgType mess, Machine m): Message* for the particular message type is declared in the *State* superclass (different processing in subclasses can then be prevented by declaring the method final).
- *Identical processing of all message in a particular state (a row in Figure 3):* A method *process(Message mess, Machine m): Message* for the *Message* superclass is declared in the particular state.

- *Known but Unexpected Messages* (denoted as “*” in

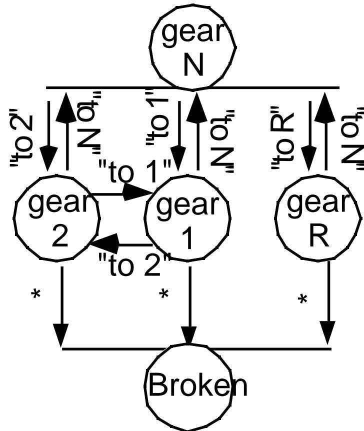

Figure 2)*:* The superclass defines that all known but unexpected messages are handled by the method “*unexpectedMsg()”.* If a given state wants to handle all unexpected messages differently, it may overload this method. Handling of some foreseen protocol errors does not involve unexpected messages - such errors belong to the case for expected messages.

- *Unknown Messages:* For messages which are now known to the protocol at all[5], the process method “*process(Message msg, Machine m)”* for general messages handles it. In our example, they are just ignored. Naturally, a state may overload this method to handle unknown messages differently.

In our example all known messages are processed as unexpected messages in the superclass and return the *Broken* state by default (this is done in the *unexpectedMsg* method). Unknown messages are just ignored.

---

5 I.e., this message subclass has not been declared to the State superclass by defining a process(NewMsg msg) method in it but it has been defined somewhere else. If a state would define a method for a message without it being defined in the superclass, this method will not be found because the superclass processes all undeclared messages.

3.5 Design Options

We now describe some options for the pattern which enable to adapt the pattern for different applications.

3.5.1 Extending the Machine

Adding a new state to an existing machine is done by just subclassing *State* and defining it as a follow-up state of another existing state.

Adding new-defined message subclasses is done by defining a *process()* method in the superclass *State* and adding a *process()* method for this message type to all states in which this message is expected. Only messages that subclass *Message* are permitted.

3.5.2 Time-Outs

Many protocols define time-out periods on events. Time-outs are easy to add to our Java extended finite state machine: First, initialize a timer object with the time-out period and a reference to the machine (this could be a thread with a simple *Thread.sleep(n)* call). Then, after the timer reached the time-out, it sends a *MsgTimeout* message to the machine. Every state that cares for time-outs has to provide processing for time-out messages.

3.5.3 Multi-Party Protocols

In a protocol with more than two participants exchanging messages, information about the messages' origin needs to be available to enable the extended finite state machine to reply to the correct addressee. This can be solved by changing the message class as follows:

- The message superclass defines an recipient's name field in order to enable the environment using the machine to transport the message to the correct recipient. This enables a machine to address messages to specific other machines. The user of the machine then asks the message returned by the machine to what address it is to be sent to.

A disadvantage of the synchronous, message-driven approach is that the machine is only able to send messages as responses to other messages. If more powerful communication is required, the machine may run as a separate thread having communication links to all machines that deliver input:

- The environment may also register communication observers at the machine which are then used for sending and receiving messages. In this scenario, the machines may be threads which run and communicate independent from the environment.

3.6 Sample Code

Now we present code fragments that illustrate the Java implementation of the gear shifting protocol we used as an example. Our message superclass is rather simple, as we do not send many parameters to the gear box:

```
   abstract public class Message{
            // dispatch() is required in all implementations for
casting
      abstract public State dispatch(State mystate, Machine m);
}
```

It follows a typical message subclass, in this case the message that represents the action “shift to reverse”. Note the *dispatch()* method that causes the message's own processing and automatically casting to its own subclass type:

```
   public class MsgToR extends Message{
      public State dispatch(State state, Machine m)
      {return state.process(this,m);}
   }
```

The other message subclasses are constructed in analogy to *MsgToR*. Now, we construct the state superclass. The default error handlers are defined in the superclass:

```
   abstract public class State{
      // Implementing shifting to Neutral is required by programmer
      //    by declaring it abstract
      abstract public State process(MsgToN msg, Machine m);

      // Here, by default all messages are unexpected.
      // Alternatively, each msg may be handled by a different
method.
      public State process(MsgToR msg, Machine m)
      { return unexpectedMsg(msg,m); }
      public State process(MsgTo1 msg, Machine m)
      { return unexpectedMsg(msg,m); }
      public State process(MsgTo2 msg, Machine m)
      { return unexpectedMsg(msg,m); }

      // This method ignores all messages unknown to the protocol.
      public State process(Message msg, Machine m){
            System.out.println("Message "+msg+" not declared in
            State.java. Ignoring it!");
            return this;
      }

      // This code handles messages that is not taken care of
      // in a particular State instance.
      // By default, it changes to the broken state.
      public State unexpectedMsg(Message msg, Machine m) {
            System.out.println("Message "+msg+" not handled by the
current
                  state. Gearset Broken.");
            return new Broken();    }
}
```

The following code illustrates the *GearR* state. It processes the two messages that are expected. For unexpected messages, *GearR* relies on the superclass message handlers.

```
public class GearR extends State {
   public State process(MsgToR msg,Machine m){
      m.outmsg=msg;
      return this; }

   public State process(MsgToN msg,Machine m) {
      m.outmsg=msg;
      return new GearN(); }
```

```
}
```

So far, messages and states of the machine have been defined, but the machine as a framework for them is missing. It looks as follows:

```
public class Machine{
    public State state=null;
    Message outmsg=null;

    // By default, create a machine in neutral state
    public Machine()
    {           state=new GearN();      // Initialize in Neutral
    }
    // Machine processes message
    public Message process(Message msg){
          outmsg=null; // discards last message sent.
          state=msg.dispatch(state,this);
          return outmsg;
    }
}
```

Using the machine is simple: call the machine object's *process()* method with the message to be processed as an argument and get the answer in return. The following example instantiates a gearset and shifts to the reverse gear and then back to neutral:

```
Machine m= new Machine();
Message msgin= new MsgToR();  // "From the network..."
Message msgout=m.process(msgin);
Message msgin= new MsgToN();

Message msgout=m.process(msgin);
```

3.7 Efficiency

One natural question is whether an object-oriented implementation, while being robust, is not efficient. Therefore, we compared our object-oriented implementation of the gearbox with a direct implementation using nested `switch`-statements. The result on our machines (Sun Ultrasparc 10 PowerMac 7600) was that the object-oriented variant is about half as fast as the direct implementation, i.e., the additional time needed for the object-oriented variant was at most 11µs per transition.
Since this test measures the overhead of the object-oriented state-*transitions* without computations, the overhead will be negligible in any protocol doing cryptographic computations. Therefore we feel that the robustness gained is well worth the performance loss.
Note that the pattern, for robustness, a new state object is created and initialized for each transition. This seems to be inefficient but our performance comparisons have shown that this is as efficient as returning references to existing state objects which are only created once.

4 Acknowledgments

We thank Tom Beiler for initial discussions on machines in Java and for helpful remarks on the theory of objects. Furthermore, we like to thank Birgit Pfitzmann, Jan-Holger Schmidt, Michael Steiner, and Martin Wanke for valueable comments. This

work was partially supported by the project ACTS SEMPER <http://www.semper.-org>; however, it represents the view of the authors only.

5 Bibliography

AJSW_97 N. Asokan, Phillipe A. Janson, Michael Steiner, Michael Waidner: The State of the Art in Electronic Payment Systems; Computer 30/9 (1997) 28-35.

FoSc_97 Martin Fowler, Kendall Scott: UML Distilled : Applying the Standard Object Modeling Language, Addison-Wesley, 1997.

GJS_96 James Gosling, Bill Joy, and Guy Steele: Java Language Specification., section 15.11, *Method Invocation Expressions* http://java.sun.com/docs-/books/jls/html/15.doc.html#20448, published in August 1996.

HoGr_89 Dieter Hogrefe: Estelle, Lotos und SDL - Standardspezifikationssprechen für verteilte Systeme, Springer Compass, Springer-Verlag, Berlin 1989.

Lync_96 Nancy A. Lynch: Distributed Algorithms; Morgan Kaufmann, San Francisco 1996.

Neum_95 Peter G. Neumann: Computer Related Risks; Addison Wesley - ACM Press, Reading Massachusetts 1995.

PfSW_98 Birgit Pfitzmann, Matthias Schunter, Michael Waidner: Optimal Efficiency of Optimistic Contract Signing; to appear at 17th Symposium on Principles of Distributed Computing (PODC), ACM, New York 1998.

Schn_96 Bruce Schneier: Applied Cryptography: Protocols, Algorithms, and Source Code in C; John Wiley & Sons, (2nd ed.) New York 1996.

ScWa_97 Matthias Schunter, Michael Waidner: Architecture and Design of a Secure Electronic Marketplace; Joint European Networking Conference (JENC8), Edinburgh, June 1997, 712.1-712.5.