ECharts: The Concise User Manual

Version 1.3 Beta

Gregory W. Bond

Text body typeset with LaTeX. Cover composed with Inkscape.

The latest version of this document is available from `http://echarts.org`

Contents

1 Introduction **1**
- 1.1 History . . . 1
- 1.2 Why Use ECharts? . . . 2

2 Developing an ECharts Machine **3**
- 2.1 Hello World! . . . 3

3 What Can ECharts Do? **7**
- 3.1 Hierarchical Machines . . . 7
  - 3.1.1 Nested Inner Machines . . . 7
  - 3.1.2 Nested External Machines . . . 8
  - 3.1.3 Reflective Invocation . . . 9
- 3.2 Actions . . . 9
  - 3.2.1 Transition Actions . . . 9
  - 3.2.2 Entry and Exit Actions . . . 10
  - 3.2.3 Action Blocks . . . 12
- 3.3 State Machines . . . 13
  - 3.3.1 Or-State Machines . . . 13
  - 3.3.2 And-State Machines . . . 13
  - 3.3.3 Mixed-State Machines . . . 14
  - 3.3.4 Machine Constructors . . . 15
- 3.4 Transitions . . . 17
  - 3.4.1 Multi-Level Transitions . . . 17
  - 3.4.2 Fork and Join Transitions . . . 18
  - 3.4.3 Transition Guards . . . 21
- 3.5 Receiving a Message . . . 24
- 3.6 Non-Blocking Execution . . . 29
- 3.7 '*' Transitions . . . 32

3.8 Sending a Message . . . 34
3.9 External Ports . . . 36
3.9.1 Output Handlers . . . 36
3.9.2 Remote Sends . . . 38
3.10 Pseudostates . . . 38
3.10.1 DEFAULT_INITIAL . . . 38
3.10.2 TERMINAL . . . 41
3.10.3 DEEP_HISTORY . . . 44
3.10.4 NEW . . . 46
3.10.5 Summary . . . 47
3.11 Machine Arrays . . . 47
3.11.1 Implicit Machine Reference . . . 48
3.11.2 Explicit Machine Reference . . . 50
3.11.3 Related References . . . 60
3.12 Timed Transitions . . . 60
3.13 Submachine Access . . . 64
3.14 Internal Ports . . . 67
3.15 Incomplete State References . . . 68
3.16 Host Language Interface . . . 72
3.17 Machine Serialization . . . 72

4 The Runtime Model **77**
4.1 Machine Execution . . . 77
4.2 Transition Evaluation . . . 78
4.3 Transition Scheduling . . . 78
4.4 Port Priorities . . . 80
4.5 Transition Priorities . . . 81
4.5.1 Message Class Rule . . . 81
4.5.2 Source Coverage Rule . . . 83
4.5.3 Transition Depth Rule . . . 87
4.6 Message Dequeuing . . . 89
4.6.1 Explicit Message Consumption . . . 89
4.6.2 Implicit Message Deferral . . . 91
4.7 Machine Lifecycle . . . 93
4.7.1 Machine Creation . . . 93
4.7.2 Machine Destruction . . . 98
4.8 Shared Data . . . 100
4.8.1 Guard Variables . . . 100
4.8.2 Port Variables . . . 103
4.9 Machine and State Access Modifiers . . . 104

5 The Machine Runtime **105**
5.1 Initialization . . . 105
5.2 Properties File . . . 106
5.3 Startup Messages . . . 106
5.4 Transition Timer Manager . . . 106
5.5 Monitoring and Logging . . . 107
5.5.1 Monitors . . . 107
5.5.2 Events . . . 108
5.5.3 putEvent() . . . 108
5.5.4 Event Filters . . . 109
5.5.5 Formatters and Logging . . . 109
5.6 Debugging . . . 110
5.6.1 Output Format . . . 110
5.6.2 Message Properties . . . 114
5.6.3 Options . . . 115
5.6.4 Global State Output . . . 115
5.7 Options Summary . . . 117

6 Generating Diagrams **119**
6.1 Page Diagrams . . . 120
6.2 Embedded Diagrams . . . 121
6.3 Customizing the Layout . . . 122
6.3.1 Overriding `dot` Layout . . . 123
6.3.2 Overriding `ech2dot` Layout . . . 124
6.4 `dot` Layout Bugs . . . 125

7 Generating Documentation **127**
7.1 Interacting with Diagrams . . . 127
7.2 `ech2doc` . . . 128
7.3 `ech2javadoc` . . . 128
7.4 SVG Viewers . . . 133

8 Command Reference **135**
8.1 Machine Dependencies . . . 135
8.2 ECHARTSPATH . . . 135
8.3 `ech2java` . . . 136
8.4 `ech2dot` . . . 137
8.5 `ech2doc` . . . 138
8.6 `ech2javadoc` . . . 140
8.7 `javadocpp` . . . 140

9 Roadmap **143**
9.1 Exception Handling 143
9.2 Machine Inheritance 143
9.3 Machine Variables 143

A Building and Using ECharts **145**
A.1 Software Requirements 145
A.2 Building ECharts 146
A.3 Using ECharts 146
A.3.1 Direct Invocation 146
A.3.2 Indirect Invocation 147

B Licenses **149**
B.1 Common Public License v1.0 149
B.2 ANTLR 2 License 155

Bibliography **157**

Chapter 1

Introduction

What is ECharts? ECharts is a state machine-based programming language for event-driven systems derived from the standardized UML Statecharts language [6]. ECharts distinguishes itself from other Statecharts dialects by focusing on implementation issues such as determinism and code re-use [2]. Like Statecharts, ECharts supports hierarchical state machines, concurrent machines and a graphical syntax. Unlike Statecharts, ECharts supports a simple textual syntax, machine reuse, multiple transition priority levels to minimize non-determinism, machine arrays, and a new approach to inter- and intra- machine communication. ECharts is a hosted language which means that it is dependent on an underlying programming language such as Java[1]. ECharts has a proven track-record in a large-scale commercial deployment. ECharts is available as open source under the Common Public License Version 1.0 (see Appendix B). Take a look at what ECharts has to offer!

1.1 History

ECharts was originally developed at AT&T Labs–Research to support an advanced telecommunications project which evolved into a nation-wide commercial product offering [3]. Currently ECharts is used in a number of new research projects at AT&T.

[1] Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. echarts.org is independent of Sun Microsystems, Inc.

1.2 Why Use ECharts?

An ECharts machine is a specification of event-driven behavior. Application domains that will benefit from using ECharts are telecommunications, web applications, user interfaces, . . . really any application domain where a system responds to events occurring in its environment.

ECharts isn't meant to be a stand-alone programming language. ECharts specifies a program's control flow in response to events received from the environment. To specify data operations, ECharts machines rely on host language statements embedded in the machine, for example, Java. Furthermore, an ECharts machine only constitutes a program unit in a larger host language program, it isn't necessarily a program's locus of control.

Chapter 2

Developing an ECharts Machine

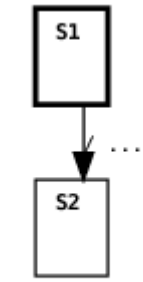
The ECharts SDK (System Development Kit) is split into two major components: (1) a translator and (2) a runtime system. The translator translates an ECharts machine into a host language program module, for example, Java. The runtime system is a host language library comprising the ECharts machine interpreter, that is, the code that actually executes the machine. The runtime system library is linked with the compiled ECharts machine modules and other compiled host language modules to form an executable program. For information on building the ECharts SDK see Appendix A.

2.1 Hello World!

Here's an example of how to program, compile, and run the canonical "Hello World!" program as an ECharts machine. In this example and all subsequent examples, we employ Java as the ECharts host language.

First, create the ECharts machine itself in a file *examples/Example0001.ech*. If you feel like cheating a bit, you'll find this file and all other example files mentioned in this document in the *runtime/java/src/examples* directory of the ECharts SDK.

```
package examples;

public machine Example0001 {
    initial state S1;
    state S2;
    transition S1 - / System.out.println("Hello World!") -> S2;
}
```



Example0001

For all examples in this document we depict a program using the ECharts textual syntax and its graphical syntax. For a discussion of the many ways that exist to generate machine diagrams like the one above see Section 6.

The `Example0001` machine does nothing more than transition from state `S1`, declared as the machine's `initial` state, to state `S2` executing the action to print "Hello World!" in the process. The graphical syntax depicts the initial state with a bold outline. In ECharts, transition actions are defined following the `/` (forward slash) character. Here the action is a simple expression, `System.out.println("Hello World!")`, specified using Java syntax. In general, Java expression syntax can be used to specify machine actions regardless of the host language. These are translated to the appropriate host language expressions (in this case, Java expressions) appropriately. ECharts also permits otherwise unacceptable expressions to be wrapped in host code delimiters `<* *>`. Further discussion concerning the ECharts/host language interface can be found in Section 3.16.

To translate the machine to the host language, invoke the ECharts translator (provided with the ECharts SDK) from the *examples* directory. In this example and subsequent examples, we show a command line prompt as a percent character `%`.

```
% ech2java --echartspath .. Example0001.ech
```

For information on configuring your platform to run ECharts commands like `ech2java` see Appendix A. This command will create a file *examples/Example0001.java*. Note the use of the `--echartspath` command line option. This option informs the translator where to locate the `examples` package declared in our machine. It is analogous to Java's `javac -classpath` command line option. See Section 8 for more `ech2java` command line options.

To compile and link the translated machine, invoke the host language's compiler from the *examples* directory:

```
% javac -classpath ..:../../echarts.jar Example0001.java
```

This command creates the file *examples/Example0001.class*. Note that the `-classpath` option references the *echarts.jar* file containing the ECharts Java runtime system (part of the ECharts SDK build).

The next step is to write a host language program whose mainline procedure calls the machine. To do this create a file *examples/HelloWorld.java*:

```
package examples;

public class HelloWorld {
    public static final void main(String argv[]) {
        try {
            new Example0001().run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

All this program does is create a new instance of an Example0001 machine and then invoke its `run()` method. The `run()` method returns after the ECharts machine runtime has determined that it can execute no further machine transitions.

Now, from the *examples* directory, compile and link the mainline program:

```
% javac -classpath ..:../echarts.jar HelloWorld.java
```

This will produce the file *examples/HelloWorld.class*. As the final step, run the program:

```
% java -classpath ..:../echarts.jar examples.HelloWorld
Hello World!
```

There you have it. Naturally, the steps described above for building an executable program can be automated using your favorite build tool such as ant or make. A GNU Makefile and and Apache Ant build.xml file are included in ECharts SDK for building the example machines in this document. For more information about ECharts runtimes and translators see Sections 5 and 8, respectively.

Chapter 3

What Can ECharts Do?

Now that you've seen how to write and run a simple ECharts program, we're ready to take a look a closer look at the ECharts language and what it can do. In the following sub-sections we provide a brief overview of ECharts language features.

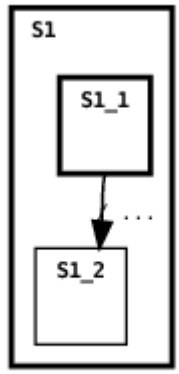
3.1 Hierarchical Machines

ECharts permits nesting a machine in a machine's state. ECharts supports two approaches to nesting machines: (1) inner machines and (2) external machines.

3.1.1 Nested Inner Machines

Here's a simple example of a nested inner machine:

```
package examples;

public machine Example0002 {
    initial state S1: {
        initial state S1_1;
        state S1_2;
        transition S1_1 - /
            System.out.println("Hello World")
        -> S1_2;
    }
}
```



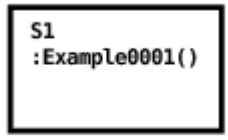
Example0002

In this example, an inner machine is nested in the initial state `S1` of the root (or top-level) machine, `Example0002`. Variables and methods defined in the parent machine are accessible to nested inner machines.

3.1.2 Nested External Machines

ECharts also supports nesting parameterized externally defined machines. This, in turn, supports machine re-use. Here's an example:

```
package examples;

public machine Example0003 {
    initial state S1: Example0001();
}
```



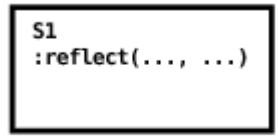
Example0003

In this example, the Example0001 machine is nested in state `S1` of the Example0003 machine. We discuss parameterized nested machines in Section 3.3.4. Access permissions for external machines are discussed in Section 4.9.

3.1.3 Reflective Invocation

In the previous example, the nested machine's identity is declared as a constant. Using ECharts reflective invocation, the nested machine can also be declared as a variable whose value is the machine's fully qualified class name as shown in this example:

```
package examples;

public machine Example0004 {

    <* static final String machineName =
       "examples.Example0001" *>
    <* static final Object[] machineArguments = null *>

    initial state S1: reflect(machineName, machineArguments);
}
```



Example0004

This example also shows how host language statements are wrapped in host language delimiters <* *>. When Java is the host language, variable declarations, method declarations and inner class declarations must be wrapped in host language delimiters. For more details concerning the ECharts/host language interface see Section 3.16.

3.2 Actions

There are four kinds of action that may execute during a machine's execution: (1) a transition action, (2) an entry action, (3) an exit action, (4) a constructor.

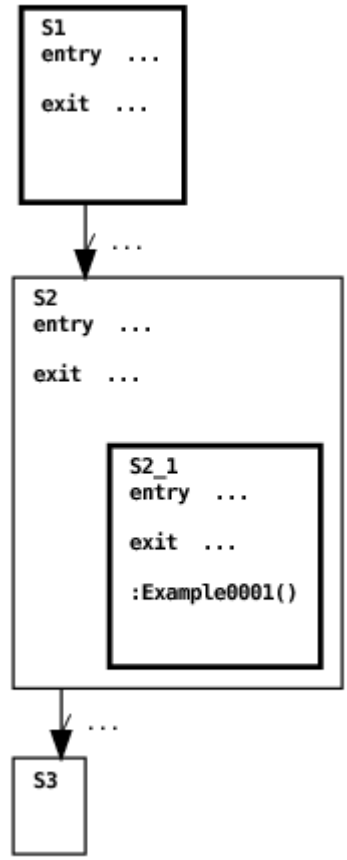
3.2.1 Transition Actions

As we've seen in the examples above, an ECharts machine can perform actions when its machine transitions fire.

3.2.2 Entry and Exit Actions

A machine can also perform actions upon entry to or exit from a machine's state. A state's exit action is executed when a firing transition explicitly references the state as a source state. A state's entry action is executed when a firing transition explicitly references the state as a target state. For example:

```
package examples;

public machine Example0005 {
    initial state S1
        entry System.out.println("Entering S1")
        exit System.out.println("Exiting S1");
    state S2
        entry System.out.println("Entering S2")
        exit System.out.println("Exiting S2") : {
        initial state S2_1
            entry System.out.println("Entering S2_1")
            exit System.out.println("Exiting S2_1") :
            Example0001();
    }
    state S3;
    transition S1 - / System.out.println("Hello") -> S2;
    transition S2 - / System.out.println("World!") -> S3;
}
```



This program produces the following output:

```
Exiting S1
Hello
Entering S2
Hello World!
Exiting S2
World!
```

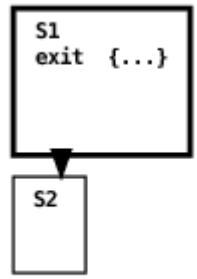
When the first transition fires, its source explicitly references state `S1` which results in its exit action being executed. The same transition's target explicitly references state `S2` which results in its entry action being executed. When the second transition fires, its source explicitly references state `S2` which results in its exit action executing. Notice that the entry action for

S1 and the entry and exit actions for S2_1 are not executed because neither of these two states are explicitly referenced by transition source or target. Entry actions execute top-down as a new machine state is entered. Exit actions execute bottom-up as an old machine state is exited. More details concerning entry and exit actions are discussed in the section devoted to machine pseudostates in Section 3.10.

3.2.3 Action Blocks

In the previous example, only single line action statements were used. Actions can also be grouped together into multi-line action blocks as follows:

```
package examples;

public machine Example0006 {
    initial state S1
        exit {
          System.out.print("Exiting ");
          System.out.print("from ");
          System.out.println("S1");
        }
    state S2;
    transition S1 --> S2;
}
```



Example0006

As you can see from this example, action blocks are enclosed by curly brackets { } and individual actions are delimited with a semicolon.

3.3 State Machines

In this section we discuss the different types of ECharts machines and states and how they are defined.

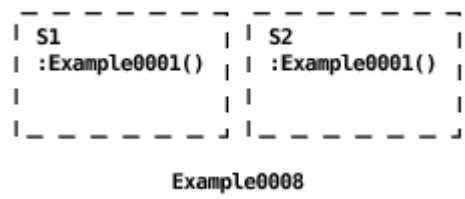
3.3.1 Or-State Machines

So far, all our example machines have been instances of *or-state machines*. An or-state machine consists of *or-states*, exactly one of which is the machine's current state at any given time. For this reason an or-state machine corresponds to the traditional notion of a state machine.

3.3.2 And-State Machines

The states of an *and-state machine* are *and-states*. Unlike an or-state machine, all states of an and-state machine are current states. As such, and-state machines support concurrent machine operation. Here's an example:

```
package examples;

public concurrent machine Example0008 {
    state S1: Example0001();
    state S2: Example0001();
}
```



This machine concurrently runs two instances of the Example0001 machine. And-states are graphically depicted with dashed lines. Here's the machine's output:

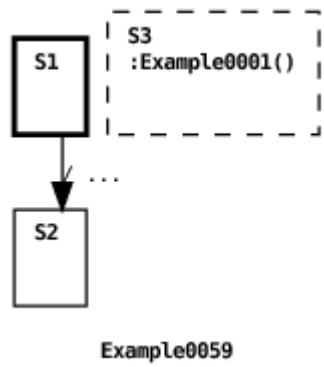
```
Hello World!
Hello World!
```

The machine declaration includes the `concurrent` machine modifier, marking it as an and-state machine. Inner submachines may also be declared as concurrent. Since both of the machine's states are current states there is no need to declare a state with the `initial` state modifier.

3.3.3 Mixed-State Machines

As its name implies, a *mixed-state machine* contains both or-states and and-states. In fact, or-state machines and and-state machines are just special cases of mixed-state machines. Here's an example of a mixed-state machine.

```
package examples;

public machine Example0059 {

    initial state S1;
    state S2;
    concurrent state S3: Example0001();

    transition S1 - / System.out.println("Hola Mundo!") -> S2;
}
```



And here's one of two possible machine outputs:

```
Hello World!
Hola Mundo!
```

As shown in the example, individual and-states in a mixed state machine, like state `S3`, are declared as `concurrent` (and or-states aren't). In the example, the top-level machine transitions from the initial or-state `S1` to or-state `S2`, thereby changing the current or-state from `S1` to `S2`. However, the and-state `S3` is always current, regardless of the current or-state. State `S3` nests the simple "Hello World!" machine discussed in Section 2.1. Because `S3` is an and-state, its submachine can execute concurrently with

the transition occurring between the or-states `S1` and `S2`. This is why the machine has two possible outputs depending on which transition fires first.

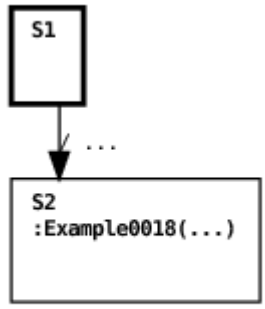
In light of this discussion you will recognize that an or-state machine is just a mixed-state machine consisting only of or-states, and an and-state machine is a mixed-state machine consisting only of and-states. It follows that the `concurrent` machine modifier, used to declare a machine as an and-state machine, is just a short-hand for declaring all the states of the machine to be `concurrent`. Like or-state machines and and-state machines, mixed-state machines can be used as internal or external submachines.

For more insight into how concurrently executing states operate see Section 4.3. Also, in Section 4.8 we discuss data sharing amongst concurrent submachines. In Sections 4.5.1 and 4.8 we discuss in detail how transitions are chosen to fire across concurrent submachines.

3.3.4 Machine Constructors

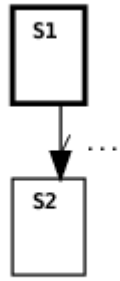
ECharts machines may declare multiple constructors if desired. By default, all machines implicitly declare a zero-argument constructor. This has been the case for all the examples to this point. The next example shows how to declare a machine constructor and how to invoke the constructor from another machine.

```
package examples;

public machine Example0017 {
    <* private String outString = null *>
    initial state S1;
    state S2: Example0018(outString);
    transition S1 - / outString = "Hello World!" -> S2;
}
```



Example0017

```
package examples;

public machine Example0018 {

    <* final private String outString *>

    public Example0018(String outString) {
        this.outString = outString;
    }
    initial state S1;
    state S2;
    transition S1 - / System.out.println(outString) -> S2;
}
```



Example0018

The `Example0017` machine invokes the `Example0018` machine when it transitions from `S1` to `S2`. When the transition fires the transition action initializes the value of the `Example0018` machine constructor parameter. The

parameter value is referenced by the `Example0018` machine constructor when its own transition fires, resulting in the following output:

```
Hello World!
```

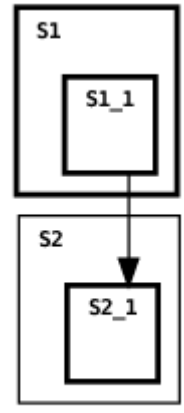
3.4 Transitions

This section describes the different forms that state transitions may take.

3.4.1 Multi-Level Transitions

In the previous examples, transitions have referenced states in the same machines the transitions are declared in. ECharts also supports transitions referencing states declared in submachines, as shown in this example:

```
package examples;

public machine Example0007 {

    initial state S1: {
        initial state S1_1;
    }
    state S2: {
        initial state S2_1;
    }
    transition S1.S1_1 --> S2.S2_1;
}
```



Example0007

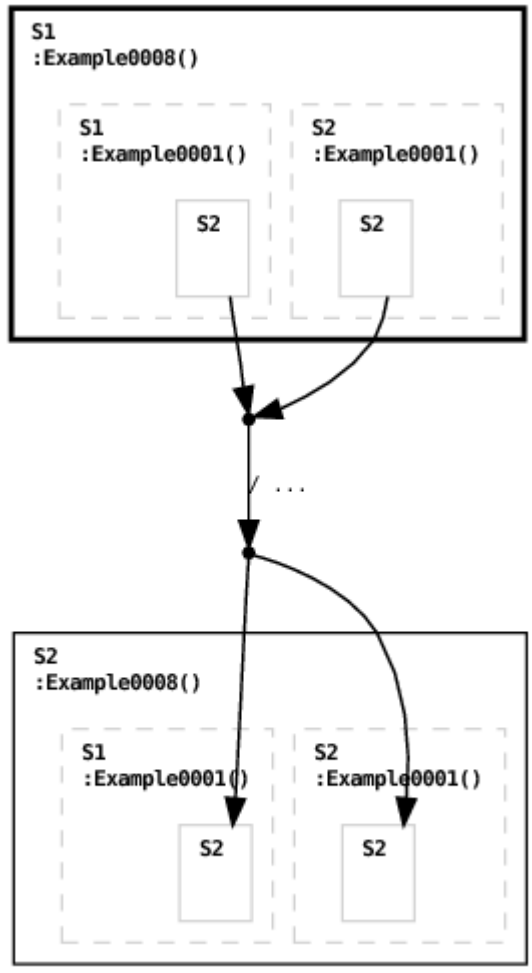
Here, the transition source references state `S1_1` of the submachine nested in `S1`, and the transition target references state `S2_1` of the submachine nested in state `S2`.

Machine state access permissions can be specified to dictate whether a state can be referenced by a transition. See Section 4.9 for further discussion.

3.4.2 Fork and Join Transitions

When mixed-state or and-state machines are used, the programmer may want a transition to reference more than one concurrent state. To do this ECharts supports the concept of *fork* and *join* transitions. Here's an example:

```
package examples;

public machine Example0009 {
    initial state S1: Example0008();
    state S2: Example0008();
    transition [ S1.S1.S2, S1.S2.S2 ] - /
        System.out.println("Hola Mundo!") ->
    [ S2.S1.S2, S2.S2.S2 ];
}
```



Example0009

The graphical syntax depicts fork and join points with small filled circles. States of external submachines that are explicitly referenced by a machine's transitions are graphically depicted with gray borders. If an external machine's states aren't explicitly referenced then they are not shown in order to reduce clutter.

Here's the machine's output:

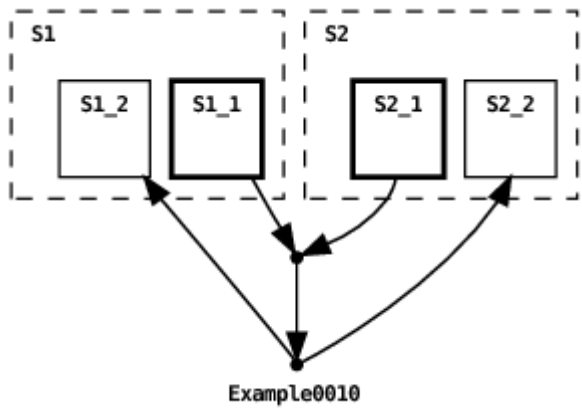
```
Hello World!
Hello World!
Hola Mundo!
Hello World!
```

```
Hello World!
```

The source states referenced by the (join) transition in the root machine `Example0009` are sub-states of the and-state machine `Example0008` nested in `S1`. Similarly the target states referenced by the (fork) transition are sub-states of the and-state machine `Example0008` nested in `S2`.

The previous example illustrated the use of fork/join transitions in a top-level or-state machine. Here's an example using them for a top-level and-state machine.

```
package examples;

public concurrent machine Example0010 {
     state S1: {
         initial state S1_1;
         state S1_2;
     }
     state S2: {
         initial state S2_1;
         state S2_2;
     }
     transition [ S1.S1_1, S2.S2_1 ] --> [ S1.S1_2, S2.S2_2 ];
}
```

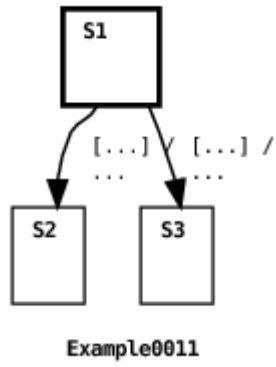


We discuss mixed-state and and-state machine transitions further in Section 3.15.

3.4.3 Transition Guards

A predicate, called a guard, can be specified for a transition. In order for a transition to fire it is necessary for the transition's guard to evaluate to true. For example:

```
package examples;

public machine Example0011 {
    <* static final int i = 0 *>
    initial state S1;
    state S2;
    state S3;
    transition S1 - [ i == 0 ] /
        System.out.println("i == 0") -> S2;
    transition S1 - [ i > 0 ] /
        System.out.println("i > 0") -> S3;
}
```



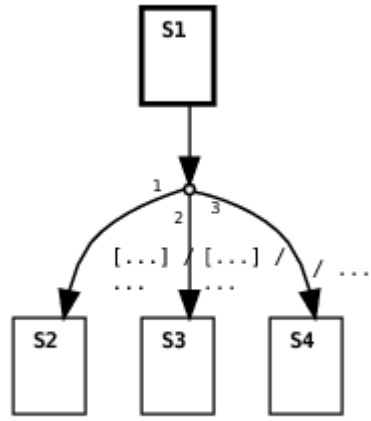
Example0011

Because `i` is initialized to `0`, only the first transition will be enabled so this machine will transition to state `S2`.

Guards can also be grouped into if-elif-else style compound statements as shown in the following example:

```
package examples;

public machine Example0049 {

    <*
```

```
    private int expensiveOperation() {
        return 42;
    }
    *>
    <* private int result = -1 *>

    initial state S1;
    state S2;
    state S3;
    state S4;

    transition S1 - [ (result = expensiveOperation()) < 42 ] /
        System.out.println("Answer is less than 42")
    -> S2
    else [ result > 42 ] /
        System.out.println("Answer is greater than 42")
    -> S3
    else  /
        System.out.println("Answer is 42!")
    -> S4;
}
```



Example0049

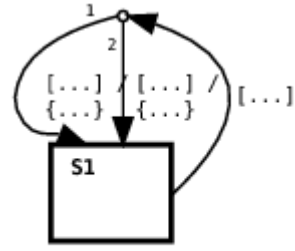
The graphical syntax depicts branching points as small hollow circles and branch segments include branch numbers to indicate guard evaluation order. And here's the output from the example:

```
Answer is 42!
```

In this example there is a single transition with three possible targets. The resulting target depends on which of the three guards evaluates to true. The guards are evaluated in the order they are written just as they are in other languages such as Java. This permits minimizing the number of calls to expensive operations that may be invoked when evaluating guards. In the example, the first guard condition is the only one that invokes the `expensiveOperation()` method. This guard has the side-effect of initializing the value of the `result` variable to the result returned by the method. Subsequent guard conditions reference the `result` value instead of re-invoking the method. Note that in this example we wrap host language method and variable declarations in host language delimiters. For more details concerning the ECharts/host language interface see Section 3.16.

Here's an example showing how compound targets can be nested.

```
package examples;

public machine Example0045 {
    <* private int iterations = 0 *>
    <* static final int MAX_ITERATIONS = 2 *>
    initial state S1;
    transition S1 -
        [ iterations < MAX_ITERATIONS ] {
            [ iterations == 0 ] / {
                System.out.println("iteration: 1");
                iterations++;
            } -> S1
            else [ iterations == 1 ] / {
                System.out.println("iteration: 2");
                iterations++;
            } -> S1
        }
}
```



Example0045

And here's the output from the example:

```
iteration: 1
iteration: 2
```

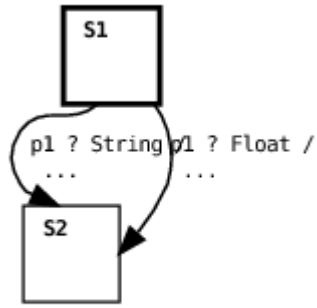
In this example, there is a single transition with nested compound targets. The machine iterates two times from state `S1` to state `S1`, each time printing out the iteration number.

We discuss guards further in Section 3.5.

3.5 Receiving a Message

All the transitions in the examples discussed so far have been *messageless* transitions. We now introduce *message* transitions. An ECharts machine reacts to its environment when it receives messages on ports specified by its message transitions. ECharts defines two classes of ports to receive messages on: external ports and internal ports. External ports support interaction with the environment outside a machine (see Section 3.9), and internal ports support interaction within a machine (see Section 3.14). In this section we discuss receiving messages from the external environment. Consider the following example machine:

```
package examples;

public machine Example0012 {

    <* final private ExternalPort p1 *>

    public Example0012(ExternalPort p1) {
        this.p1 = p1;
    }
    initial state S1;
    state S2;
    transition S1 - p1 ? String /
        System.out.println("string: " + message)
    -> S2;
    transition S1 - p1 ? Float /
        System.out.println("float: " + message)
    -> S2;
}
```



Example0012

Here's the machine's environment:

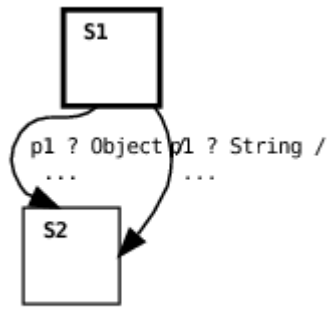
```
package examples;

import org.echarts.ExternalPort;
import org.echarts.MachineThread;

public class Example0012Environment {

    final static private ExternalPort p1 =
        new ExternalPort("p1");

    final static public void main(String[] argv) {
        try {
            new MachineThread(new Example0012(p1)).start();
            p1.input("Hello World!");
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

First, the machine's environment creates an instance of the machine and runs it on a separate thread. The `MachineThread` class is a convenience class that does nothing more than call the machine's `run()` method. The machine's thread blocks waiting for a message to arrive on `p1`. Then the machine's environment puts a message, a `String` instance, in external port `p1`'s input queue. The arrival of the message unblocks the machine's thread, and the transition specifying a `String` message fires, printing out the message. ECharts also supports non-blocking machine execution, described in Section 3.6.

A message transition uses the port receive syntax `p1 ? Classname` to specify waiting on external port `p1` for a message that is an instance of the `Classname` class. When the transition's guard is evaluated and when the transition's action is executed, the message instance is made available as a distinguished variable named `message`. The port instance is made available as a distinguished variable named `port`. While this variable's value is redundant here, it is useful for '*' transitions (see Section 3.7).

When enabled message transitions specify the same port and same received message, then the ECharts runtime will give highest priority to the transition specifying the most specific message class. Here's an example:

```
package examples;

public machine Example0047 {

    <* final private ExternalPort p1 *>

    public Example0047(ExternalPort p1) {
        this.p1 = p1;
    }

    initial state S1;
    state S2;
    transition S1 - p1 ? Object /
        System.out.println("Object: " + message)
    -> S2;
    transition S1 - p1 ? String /
        System.out.println("String: " + message)
    -> S2;
}
```



Example0047

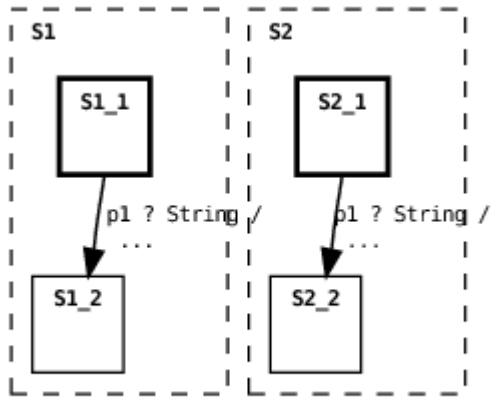
```
package examples;

import org.echarts.ExternalPort;

public class Example0047Environment {

    public static final void main(String[] argv) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
            p1.input("Hello World!");
            new Example0047(p1).run();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

This example shows the machine's `run()` method invoked directly by the environment's `main()` method as an alternative to creating a separate `MachineThread` instance as shown in the previous example. When the machine is executed the runtime chooses the second transition to fire because it specifies the `String` class which is more specific than the `Object` class (which is a superclass of the `String` class). This transition priority rule is the first of three transition priority rules supported by the ECharts runtime. These rules are discussed in more detail in Section 4.5.

A given external port can be referenced by more than one transition in a machine, for example, by transitions in different concurrent submachines of an and-state machine:

```
package examples;

public concurrent machine Example0013 {

    <* final private ExternalPort p1 *>

    public Example0013(ExternalPort p1) {
        this.p1 = p1;
    }
    state S1: {
        initial state S1_1;
        state S1_2;
```

```
        transition S1_1 - p1 ? String /
            System.out.println(message)
        -> S1_2;
    }
    state S2: {
        initial state S2_1;
        state S2_2;
        transition S2_1 - p1 ? String /
            System.out.println(message)
        -> S2_2;
    }
}
```



Example0013

Here's the previous machine's environment:

```
package examples;

import org.echarts.ExternalPort;
import org.echarts.MachineThread;

public class Example0013Environment {

    public static final void main(String[] argv) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
            new MachineThread(new Example0013(p1)).start();
            p1.input("Hello");
            p1.input("World!");
```

```
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

In this example, the machine's environment puts two messages in `p1`'s input queue. The two transitions in the machine's concurrent states fire in succession producing one of two possible outputs. Here's one possible output:

```
Hello
World!
```

Normally, a port variable's value will remain constant throughout the life of a machine. However, if a port's value is expected to change then there are constraints on how that value may be changed. This is discussed in more detail in Section 4.8.2.

We discuss the blocking machine execution cycle in more detail in Section 4.1. A non-blocking alternative is discussed in the next section, Section 3.6. We discuss how messages received from the environment are dequeued in more detail in Section 4.6.

3.6 Non-Blocking Execution

In the last two examples we discussed how the thread that invokes a machine's `run()` method can block while awaiting the arrival of messages specified by message transitions in the machine's current state. ECharts also provides a non-blocking `run()` method that can be useful in some application domains such as HTTP or SIP servlet containers. Here's how the machine in the previous example would be executed using a non-blocking approach:

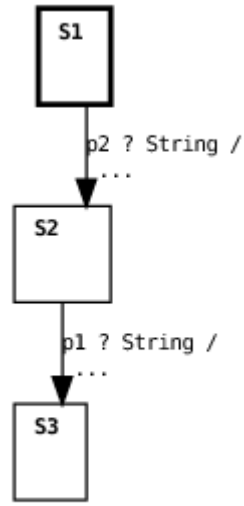
```
package examples;

import org.echarts.ExternalPort;
import org.echarts.TransitionMachine;

public class Example0014Environment {

    public static final void main(String[] argv) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
```

```
            TransitionMachine machine = new Example0014(p1);
            machine.run(p1, "Hello");
            machine.run(p1, "World!");
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

Here, the machine's environment creates a machine instance (where `Machine0014` is identical to `Machine0013`) and then invokes the non-blocking `run()` method two times. The arguments for the `run()` method specify the port `p1` and the message associated with the port.

In the case where the non-blocking `run()` method is invoked specifying port p and no active message transitions specify p in the machine's current state then the message will be enqueued on p's input queue in accordance with the implicit message deferral model supported by the ECharts runtime (see Section 4.6.2). Messages enqueued as a result of earlier `run()` invocations may be dequeued and processed by later `run()` invocations in accordance with the runtime's explicit message consumption model (see Section 4.6.1). Here's an example:

```
package examples;

public machine Example0048 {

    <* final private ExternalPort p1 *>
    <* final private ExternalPort p2 *>

    public Example0048(ExternalPort p1, ExternalPort p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    initial state S1;
    state S2;
    state S3;
    transition S1 - p2 ? String /
        System.out.println(port + ": " + message)
    -> S2;
    transition S2 - p1 ? String /
        System.out.println(port + ": " + message)
    -> S3;
}
```



Example0048

```
package examples;

import org.echarts.ExternalPort;

public class Example0048Environment {

    final static public void main(String[] argv) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
            final ExternalPort p2 = new ExternalPort("p2");
            final Example0048 machine =
                new Example0048(p1, p2);
            machine.run(p1, "Hello");
            machine.run(p2, "World");
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

And here's the example's output:

```
p2: World
p1: Hello
```

In this example, the environment invokes the machine's non-blocking `run()` method two times: first with a message for port `p1` and then with a message for port `p2`. Since the only message transition specifying `p1` is not active in the machine's initial state `S1`, then `p1`'s message is enqueued on `p1`'s input queue and the machine's state remains unchanged the first time `run()` is invoked. When `run()` is invoked a second time, the message transition specifying `p2` fires. Then the method dequeues `p1`'s message and fires `p1`'s transition because the transition is active in state `S2`.

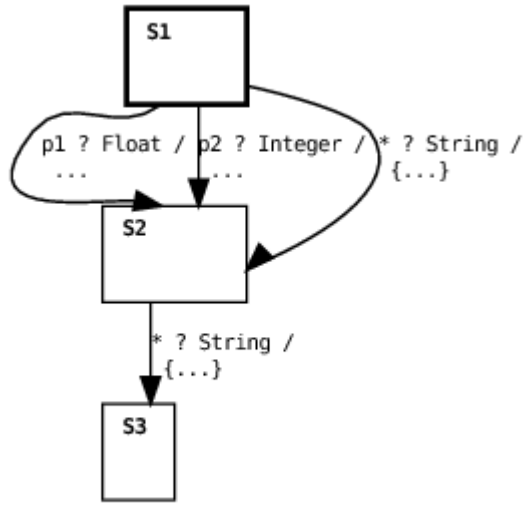
A blocking approach to machine execution is discussed in the previous section, Section 3.5.

3.7 '*' Transitions

A variation on the message transition is the '*' ("any port") transition. The '*' transition is a message transition that specifies a wildcard port. In particular, it matches any port that is explicitly mentioned in some other message transition in a machine's current state. Here's an example machine using a '*' transition:

```
package examples;

public machine Example0015 {

    <* final private ExternalPort p1 *>
    <* final private ExternalPort p2 *>

    public Example0015(ExternalPort p1, ExternalPort p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    initial state S1;
    transition S1 - p1 ? Float /
        System.out.println(message)
    -> S2;
    transition S1 - p2 ? Integer /
        System.out.println(message)
    -> S2;
    transition S1 - * ? String / {
            System.out.println(port);
            System.out.println(message);
    } -> S2;
```

```
    state S2;
    transition S2 - * ? String / {
        System.out.println(port);
        System.out.println(message);
    } -> S3;
    state S3;
}
```



Example0015

And here's the machine's environment:

```
package examples;

import org.echarts.ExternalPort;
import org.echarts.MachineThread;

public class Example0015Environment {

    public static final void main(String[] argv) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
            final ExternalPort p2 = new ExternalPort("p2");
            new MachineThread(new Example0015(p1, p2)).start();
            p1.input("Hello World!");
            p1.input("Hola Mundo!");
        } catch (Exception e) { e.printStackTrace(); }
```

```
    }
}
```

Here's the machine's output:

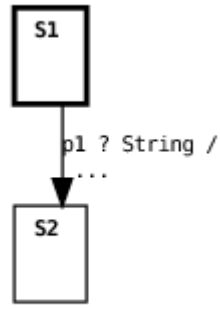
```
p1
Hello World!
```

In this example, the machine's environment puts two `String` instances in external port `p1`'s input queue. The only transition to explicitly reference `p1` in state `S1` specifies a `Float` message. However, the '*' transition in `S1` does specify a `String` instance and, since the '*' matches any port specified by a transition in the current state, it matches `p1` and fires. Notice that the '*' transition action references a distinguished variable named `port` whose value is the matched port. In state `S2`, port `p1` has one `String` message remaining in its input queue. Furthermore, there is a '*' transition defined in that state. However, since there are no transitions explicitly referencing `p1` in `S2` the '*' transition is not enabled to fire. For more information regarding message queuing see Section 4.6.

3.8 Sending a Message

Naturally, in addition to receiving messages, a machine can also send messages to its environment or to ancestor/descendant machines via external or internal ports, respectively. Here's an example:

```
package examples;

public machine Example0019 {

    <* final private ExternalPort p1 *>

    public Example0019(ExternalPort p1, String message) {
        this.p1 = p1;
        p1 ! message;
    }
    initial state S1;
    state S2;
    transition S1 - p1 ? String /
        System.out.println(message)
    -> S2;
}
```



Example0019

```
package examples;

import org.echarts.ExternalPort;
import org.echarts.MachineThread;

public class Example0019Environment {

    public static final void main(String[] argv) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
            final ExternalPort p2 = new ExternalPort("p2");
            p1.setPeer(p2);
            p2.setPeer(p1);
            new MachineThread(
                new Example0019(p1, "Hello")).start();
            new MachineThread(
                new Example0019(p2, "World")).start();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

In this example, the environment creates two external ports and sets them as each other's peers using the `setPeer()` method. Setting a port's peer means that when a message is sent on a port, the message will be placed in the peer port's input queue. (In Section 3.9 we discuss another way for the environment to receive messages from a machine.) Next, two instances of the `Example0019` machine are created and run on separate threads. Each machine's constructor is invoked with different values for its port and

message. Executing the machine constructor sends the specified message on the specified port which enqueues the message on the peer port's input queue. The syntax for a message send operation is `port ! message`. Since port input queue size is unbounded the port send operation is non-blocking. The port send operation can be used anywhere actions are permitted i.e. in state entry/exit actions, transition actions, or machine constructors. Finally, each machine waits for a message to arrive on its own input queue. When the message arrives, its transition fires and the message is printed. One possible output from this example is:

```
World
Hello
```

We discuss how messages received from the environment are dequeued in more detail in Section 4.6.

3.9 External Ports

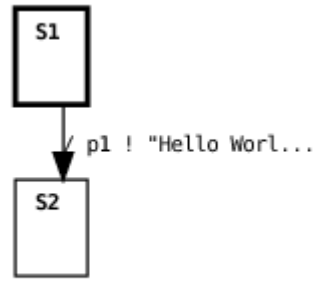
As we have seen in the previous examples, an external port maintains a input queue of messages that it has received but has not yet processed. To send messages on a port, the programmer must configure the port to output the messages to a particular destination.

3.9.1 Output Handlers

`Example0019` showed that in order to enqueue messages sent out on a port, the port can be peered with another port. However, queuing output messages is not always necessary. Instead, it may be desirable to simply invoke a listener method in the environment when a machine outputs a message. This can be accomplished by overriding the external port's `output()` method as shown in the next example:

```
package examples;

public machine Example0020 {

    <* final private ExternalPort p1 *>

    public Example0020(ExternalPort p1) {
        this.p1 = p1;
    }
    initial state S1;
```

```
    state S2;
    transition S1 - / p1 ! "Hello World!" -> S2;
}
```



Example0020

```
package examples;

import org.echarts.ExternalPort;
import org.echarts.Machine;

public class Example0020Environment {

    public static final void main(String[] argv) {
        try {
            final ExternalPort p1 = new ExternalPort("p1") {
                    final public void output(final Object message,
                                             final Machine machine)
                        throws Exception {
                        System.out.println(message);
                    }
                };
            new Example0020(p1).run();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

In this example the machine's environment creates an instance of an (anonymous) `ExternalPort` subclass. The subclass specifies that the `output()` method should simply print any message sent on that port, which is precisely what happens when the `Example0020` machine fires its transition.

3.9.2 Remote Sends

Messages may also be sent to remote external port instances. For example, imagine that the two `Example0020` machine instances in the previous example are running on different machines. The `ExternalPort` class supports this via its `getRemote()` method. This method returns a remote reference to the port. When a message is input to the remote port reference, either directly by calling the port's `input()` method, or indirectly by invoking a message send operation `!` on the port's peer, the message is placed in the actual port's input queue. For the Java runtime system this is accomplished via Java's RMI.

3.10 Pseudostates

A small number of pseudostates are defined for all machines. Unlike a real state, a pseudostate represents a machine property. Pseudostates can be included in a transition's source or target state references.

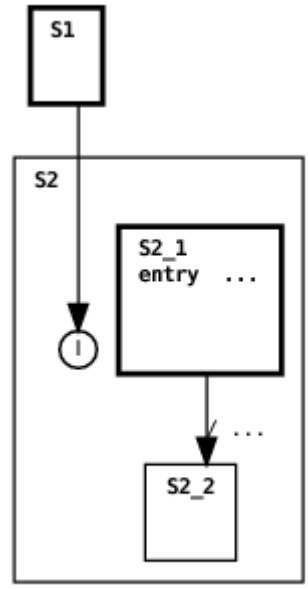
3.10.1 DEFAULT_INITIAL

The `DEFAULT_INITIAL` pseudostate references a machine's declared initial state. This pseudostate can only be referred to as part of a transition's target state reference. If the machine is an or-state or mixed-state machine that has not declared an initial state, then referencing its `DEFAULT_INITIAL` pseudostate results in a translator error.

A reference to a machine's `DEFAULT_INITIAL` pseudostate is treated as an explicit reference to the machine's initial state. Therefore, as discussed in Section 3.2.2, if an entry action is defined for the initial state then it will be executed. Here's a simple or-state machine example using the `DEFAULT_INITIAL` pseudostate:

```
package examples;

public machine Example0023 {
    initial state S1;
    state S2: {
        initial state S2_1
            entry System.out.println("Hola Mundo");
        state S2_2;
        transition S2_1 - /
            System.out.println("Hello World")
        -> S2_2;
```

```
    }
    transition S1 --> S2.DEFAULT_INITIAL;
}
```



Example0023

When the parent transition fires it references the `DEFAULT_INITIAL` pseudostate of the submachine defined in state `S2`. This pseudostate is graphically depicted by the letter 'I' enclosed by a circle. Since this is treated as an explicit reference to the submachine's initial state, the entry action defined for the initial or-state `S2_1` is executed followed by the firing of the submachine's transition. The resulting output is:

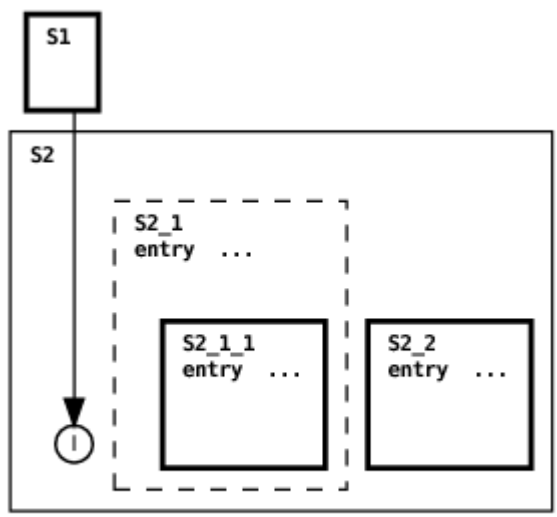
```
Hola Mundo
Hello World
```

A `DEFAULT_INITIAL` pseudostate reference to a machine array references the default initial states of all machine array elements. This is consistent with the rules for transition target state references discussed in Section 3.11.

If a machine possesses and-states, then referencing the machine's `DEFAULT_INITIAL` pseudostate is a short-hand for referencing the initial or-states of the and-state's submachines. While the interplay between the `DEFAULT_INITIAL` pseudostate and entry actions is straightforward for or-state machines, it is more subtle for mixed-state and and-state machines. If the

DEFAULT_INITIAL pseudostate of a mixed-state or and-state machine is referenced, then entry actions defined for the machine's initial or-state and and-states will be executed, however, entry actions defined for the initial states of the and-states' submachines will not be executed. This is shown in the following example with a mixed-state machine:

```
package examples;

public machine Example0060 {

    initial state S1;
    state S2: {

        concurrent state S2_1
            entry System.out.println("Hello World"): {

            initial state S2_1_1
                entry System.out.println("Hola Mundo");
        }
        initial state S2_2
            entry System.out.println("Bonjour le Monde");
    }
    transition S1 --> S2.DEFAULT_INITIAL;
}
```



Example0060

One of two possible outputs from this example is:

```
Bonjour le Monde
Hello World
```

The transition target references the DEFAULT_INITIAL pseudostate of state S2's submachine. When the transition fires, the entry actions for the submachine's and-state S2_1 and initial or-state S2_2 execute. Since these two states are entered concurrently, the order their entry actions execute is nondeterministic so it is possible for the machine output to be reversed from that shown above. However, the entry action for the initial state of the submachine nested in state S2_1 is not executed because it not explicitly referenced by the DEFAULT_INITIAL pseudostate.

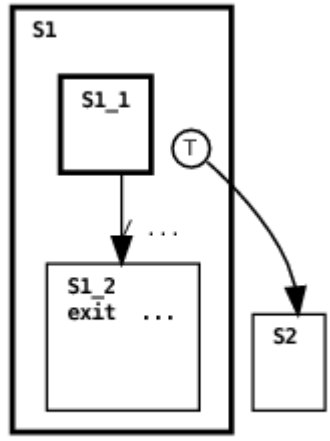
3.10.2 TERMINAL

The TERMINAL pseudostate can only be referred to as part of a transition's source state reference. A TERMINAL source state reference is satisfied if the referenced machine is in a *terminal* state. An or-state machine is defined to be in a terminal state if the machine's current or-state is a state with no submachines and no outbound transitions. An and-state machine is in a terminal state if all of its and-state submachines are in terminal states. A mixed-state machine is in a terminal state if its current or-state is terminal and the submachines of all its and-states are in terminal states.

If a TERMINAL pseudostate references a machine array, then the reference is satisfied if at least one of the machine array elements is in a terminal state. This is consistent with the rules for transition source state references discussed in Section 3.11. Similar to what was discussed for the DEFAULT_INITIAL pseudostate in Section 3.10.1, a TERMINAL pseudostate reference is considered to be an explicit reference to a machine's terminal state. Therefore, any exit actions defined for a terminal state referenced with the TERMINAL pseudostate will be executed. Here's a simple example:

```
package examples;

public machine Example0024 {
    initial state S1: {
        initial state S1_1;
        state S1_2
            exit System.out.println("Hola Mundo");
        transition S1_1 - /
```

```
            System.out.println("Hello World")
        -> S1_2;
    }
    state S2;
    transition S1.TERMINAL --> S2;
}
```



Example0024

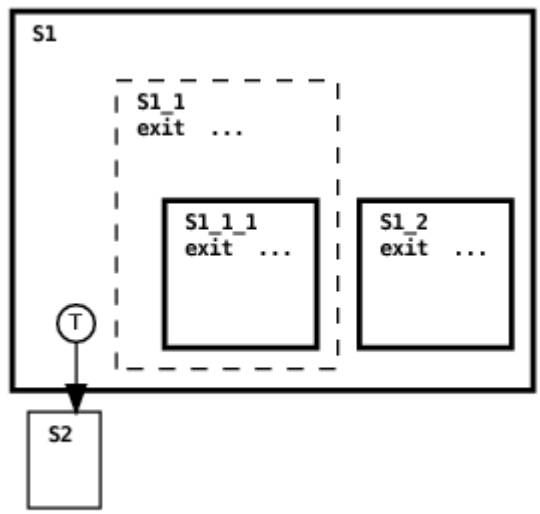
The terminal pseudostate is graphically depicted by the letter 'T' enclosed by a circle. The first thing to occur is the transition in state `S1`'s submachine fires. This results in the submachine entering a terminal orstate `S1_2`. This condition satisfies the source state reference of the parent machine's transition `S1.TERMINAL` causing the parent to transition to state `S2`. Since the reference to the `TERMINAL` pseudostate is treated as an explicit reference to `S1_2`, then `S1_2`'s exit action is executed when the parent transition fires. Here's the machine's output:

```
Hello World
Hola Mundo
```

Here's another example that demonstrates the use of the `TERMINAL` pseudostate with a mixed-state machine.

```
package examples;

public machine Example0061 {
```

```
    initial state S1: {

        concurrent state S1_1
            exit System.out.println("Hello World"): {

            initial state S1_1_1
                exit System.out.println("Hola Mundo");
        }
        initial state S1_2
            exit System.out.println("Bonjour le Monde");
    }
    state S2;
    transition S1.TERMINAL --> S2;
}
```



Example0061

One possible machine output is:

```
Bonjour le Monde
Hello World
```

The first thing to notice is that the state `S1` submachine is initially in a terminal state. This is because its initial or-state `S1_2` is a terminal state and its and-state `S1_1` is in a terminal state because its submachine is in a terminal state `S1_1_1`. This means that the `TERMINAL` pseudostate reference in the transition is satisfied and the transition fires. The next thing to

notice is that the TERMINAL pseudostate reference is interpreted to be an explicit reference to the two terminal states: S1_1 and S1_2. For this reason, the states' exit actions execute but since the two states are concurrent their exit action execution order is nondeterministic. This means another possible machine output has the opposite ordering of what is shown above. The last thing to note is that the exit action for state S_1_1 is not executed because the TERMINAL pseudostate reference is not interpreted to explicitly reference this state.

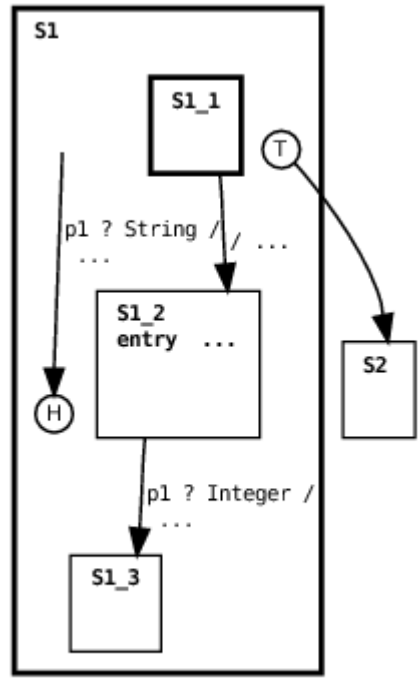
We discuss how the notion of a terminal state is related to machine destruction in Section 4.7. See Section 4.5.2 for a discussion of the relative specificity of a TERMINAL pseudostate reference in the context of the source state coverage transition priority rule.

3.10.3 DEEP_HISTORY

The DEEP_HISTORY pseudostate can only be referred to as part of a transition's target state reference. DEEP_HISTORY references a machine's current state and the current states of any of its submachines. Unlike references to the DEFAULT_INITIAL pseudostate, or the TERMINAL pseudostate discussed in Sections 3.10.1 and 3.10.2, respectively, a reference to a machine's DEEP_HISTORY pseudostate is *not* considered to be an explicit reference to the machine's current state. Therefore, any entry actions defined for the machine's current state will *not* be executed when referenced via the machine's DEEP_HISTORY pseudostate. Here's an example that should clarify this:

```
package examples;

public machine Example0025 {

    <* final private ExternalPort p1 *>

    public Example0025(ExternalPort p1) {
        this.p1 = p1;
    }
    initial state S1: {
        initial state S1_1;
        state S1_2
            entry System.out.println("Hola Mundo");
        state S1_3;
        transition S1_1 - /
            System.out.println("Hello World")
        -> S1_2;
```

```
        transition S1_2 - p1 ? Integer /
            System.out.println(message)
        -> S1_3;
    }
    state S2;
    transition S1 - p1 ? String /
        System.out.println(message)
    -> S1.DEEP_HISTORY;
    transition S1.TERMINAL --> S2;
}
```



Example0025

```
package examples;

import org.echarts.ExternalPort;

public class Example0025Environment {

    public static final void main(String[] argv) {
```

```
        try {
            final ExternalPort p1 = new ExternalPort("p1");
            p1.input("forty two");
            p1.input(new Integer(42));
            new Example0025(p1).run();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

The deep history pseudostate is graphically depicted by the letter 'H' surrounded by a circle. In this example, the machine's environment creates an external port `p1` and then enqueues two messages on the port's input queue: a string instance followed by an integer instance. Then the environment creates and runs the machine. The submachine defined in state `S1` fires its first transition from state `S1_1` to state `S1_2`, executing the entry action defined for `S1_2`, and then blocks in state `S1_2` waiting for an `Integer` instance to arrive on `p1`. Then the parent's first transition fires in response to the arrival of a `String` instance arriving on `p1` (see Section 6.4 for an explanation of the graphical depiction of this transition). The transition's target is the `DEEP_HISTORY` pseudostate of the `S1` submachine. Since the submachine was in state `S1_2` prior to the parent transition firing, then the submachine will return to state `S1_2`. Furthermore, since the `DEEP_HISTORY` reference does not constitute an explicit state reference, then the entry action defined for `S1_2` will not be executed again. Since the next message in `p1`'s input queue is an `Integer` instance, the submachine transition from `S1_2` to `S1_3` will fire. Finally, since `S1_3` is a terminal state, the parent's second transition will fire. For the record, here's the resulting output:

```
Hello World
Hola Mundo
forty two
42
```

The `DEEP_HISTORY` pseudostate will be discussed further when we discuss timed transitions in Section 3.12.

3.10.4 NEW

The `NEW` pseudostate is normally used in association with machine arrays discussed in Section 3.11. When used to reference a machine array the effect is to create a new submachine element while leaving the state of any

pre-existing machine elements undisturbed. In effect, a `NEW` pseudostate reference behaves exactly the same as a `DEEP_HISTORY` pseudostate reference for the pre-existing machine elements. Similarly, the `NEW` pseudostate behaves exactly the same as a `DEEP_HISTORY` pseudostate reference for and-state, or-state and mixed-state machines.

3.10.5 Summary

Here's a summary of how the pseudostates are interpreted by the ECharts runtime system.

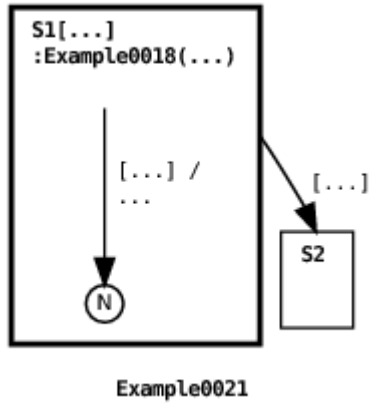
Pseudostate	Source/Target	Implicit/Explicit
DEFAULT_INITIAL	target	explicit
TERMINAL	source	explicit
DEEP_HISTORY	target	implicit
NEW	target	implicit

3.11 Machine Arrays

In addition to and-state submachines, ECharts supports another kind of concurrent submachine known as the machine array. A machine array is a bounded array of concurrent submachines (elements) nested within an and-state or an or-state. Unlike an and-state submachine, which is created when its parent machine is created, machine array elements must explicitly be created, one-by-one, by an ancestor machine at run-time.

Here's an example of machine array element creation:

```
package examples;

public machine Example0021 {

    <* private final String[] messages =
        new String[]{"Hello", "World"} *>
    <* private final int bound = messages.length *>
    <* private int created = 0 *>

    initial state S1[bound]: Example0018(messages[created - 1]);
    state S2;
    transition S1 - [ created < bound ] / created++ -> S1.NEW;
    transition S1 - [ created == bound ] -> S2;
}
```



In this example, state `S1` declares a machine array of size `bound = 2`: one element for each string element in the `messages` array. Machine array instances in `S1` are declared to be `Example0018` machines. When the `Example0021` machine is initially created it is important to realize that *no* `Example0018` instances are created. A machine instance is created only when the transition from `S1` to `S1.NEW` fires (see Section 6.4 for an explanation of the graphical depiction of this transition). The `NEW` pseudostate, introduced in Section 3.10, is graphically depicted by the letter 'N' surrounded by a circle. When then `NEW` pseudostate is referred to in the context of a machine array, the effect is to create a new machine instance in the machine array. The states of any other pre-existing machine instances in the array are unaffected. After the `Example0021` machine creates both `Example0018` submachines, it transitions to `S2`. The resulting output is:

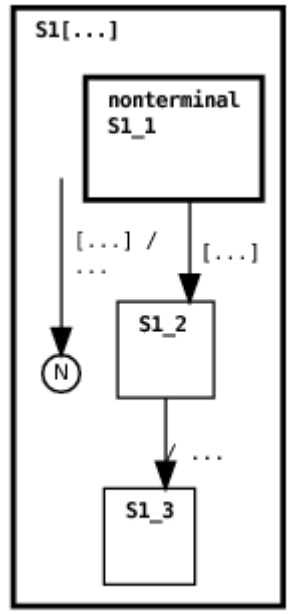
```
Hello
World
```

3.11.1 Implicit Machine Reference

When more than one machine instance exists in a machine array, you may wonder which machine is referenced when a transition references the array. There are two simple rules: (1) a transition source state reference is satisfied if *any* machine in the array satisfies the source state; (2) a target state reference refers to *all* pre-existing machines in the array. These rules are highlighted in the following example:

```
package examples;
```

```
public machine Example0022 {

    <* private final String[] messages =
        new String[]{"Hello", "World"} *>
    <* private final int bound = messages.length *>
    <* private int created = 0 *>

    initial state S1[bound]: {
        initial nonterminal state S1_1;
        state S1_2;
        state S1_3;
        transition S1_2 - /
            System.out.println(messages[getMachineIndex()])
        -> S1_3;
    }
    transition S1 - [ created < bound ] / created++ -> S1.NEW;
    transition S1.S1_1 - [ created == bound ] -> S1.S1_2;
}
```



Example0022

Like `Example0021`, this machine first creates two submachines in the machine array declared for state `S1` (see Section 6.4 for an explanation of the

graphical depiction of this transition), however, in this case the submachine is defined as an inner machine instead of an external machine. Once created, each submachine instance is prevented from transitioning from its initial state `S1_1`. (The reason for including the `nonterminal` state modifier for `S1_1` will become clear when we discuss machine destruction in Section 4.7.) Once both submachine instances are created the source state `S1.S1_1` referenced by the parent machine's second transition is satisfied (see Section 6.4 for an explanation of the graphical depiction of this transition). This is because there exists at least one submachine in state `S1_1`. In fact, both submachines happen to be in this state. When the transition fires, it updates the state of *both* submachine instances to `S1_2` enabling the submachines' second transition to fire. The `getMachineIndex()` method referenced in the transition's action returns the index of the submachine in the parent machine's machine array. So, when the submachine transitions fire, they print out the submachines' respective messages, one possible output being:

```
Hello
World
```

In general, when a machine array is referenced by a (source) join transition (see Section 3.4.2) then the interpretation is that the individual source state references may be satisfied by different machine array elements. Explicit machine references should be used in order to constrain more than one source state reference to refer to the same machine array element as explained in Section 3.11.2.5.

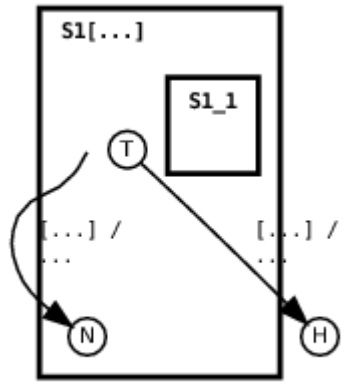
3.11.2 Explicit Machine Reference

Section 3.11.1 explains how transitions can implicitly reference machine array elements. In this section we describe how transitions can also explicitly reference particular machine array elements. This is accomplished using indexed machine array references. There are two forms of indexed reference: as a *settor* to obtain a particular array element index, or as a *gettor* to specify a particular array element index. Index settors and gettors are described in Sections 3.11.2.1 and 3.11.2.3, respectively.

3.11.2.1 Index Settors

Our first example shows the use of a settor to obtain the indices of created and destroyed machine array elements.

```
package examples;

public machine Example0052 {

    <* private final int bound = 3 *>
    <* private int created = 0 *>

    initial state S1[bound]
        entry System.out.println("created target index: " + i)
        : {    initial state S1_1;    }

    <* private int i *>

    transition S1 - [ created < bound ] / created++ -> S1[?i].NEW;

    transition S1[?i].TERMINAL - [ created == bound ] /
        System.out.println("destroyed source index: " + i)
    -> DEEP_HISTORY;
}
```



Example0052

The output from this machine is:

```
created target index: 0
created target index: 1
created target index: 2
destroyed source index: 0
destroyed source index: 1
destroyed source index: 2
```

In `Example0052`, the parent machine initially creates 3 machine array elements (see Section 6.4 for an explanation of the graphical depiction of this transition). A target index settor, expressed with the notation `S1[?i].NEW` is used to obtain the index value of an array element when it is created. The settor assigns the index value of the newly created element to the parent's `i` variable. The value obtained is then output by the state `S1` entry action.

Target index settors are only applicable to machine array element creation i.e. they are only applicable to `NEW` target state references. A target index settor assignment occurs after the execution of the associated transition's actions, but before the execution of state entry actions that may execute as a result of the transition firing. This means that the value assigned by a target index settor is not available to the associated transition's action, but it is available for reference by state entry actions. Furthermore, the value assigned to a variable by a settor persists after the associated transition fires unless, of course, the variable is explicitly assigned a new value by some other machine action or settor.

`Example0052` also illustrates the use of source index settors. In this example, the source index settor is expressed with the notation `S1[?i].TERMINAL`. This settor obtains the index value of an array element when it is in a terminal state. When the associated transition fires, the settor assigns the index value of the referenced element to the parent's `i` variable. The value obtained is then output by the transition's action.

Source index settors are applicable to any transition source state reference, not simply `TERMINAL` pseudostate references. A source index settor assignment occurs before any state exit actions execute and before the associated transition's guard or action executes. This means that the value assigned by a source index settor is available for reference by state exit actions, and it is available for reference by the associated transition's guard and action.

But now a word of caution with regards to the persistence of the value assigned by a source index settor: a side-effect of evaluating possible values for a source index settor is that the variable referenced by the settor may be overwritten during the evaluation process. For this reason, if one wants the value obtained by a source index settor to persist after its associated transition has fired then the value should be re-assigned to a variable that is not referenced by a source index settor as part of the associated transition's action.

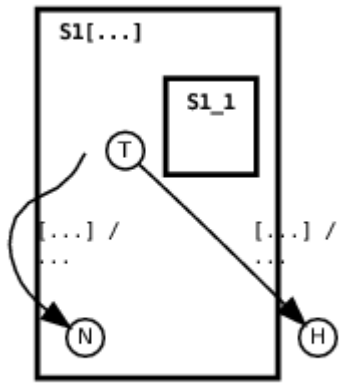
Another constraint is that each variable referenced in a source index settor should be unique for a given transition i.e. a variable shouldn't be shared across a transition's source index settors.

The use of settor-initialized variables in transition guards is discussed in more detail in Section 3.11.2.2. Other examples of source index settors are given in Sections 3.11.2.4, and 3.11.2.5, 4.8.

3.11.2.2 Source Index Settors and Guards

As mentioned in Section 3.11.2.1, an index assigned by a source index settor can be referenced in the associated transition's guard. In this case the ECharts runtime searches for an index value that both satisfies the transition's source state reference and the transition's guard. If such an index is found then the transition will be a candidate for firing. Here's an example:

```
package examples;

public machine Example0054 {

    <* final private int bound = 4 *>

    initial state S1[bound]: {
        initial state S1_1;
    }
    <* private int created = 0 *>

    transition S1 - [created < bound] / created++
    -> S1[created-1].NEW;

    <* private int i = -1 *>

     transition [ S1[?i].TERMINAL] -
         [ created == bound && (i % 2) == 0 ] /
         System.out.println("index: " + i + " terminated")
     -> DEEP_HISTORY;
}
```



Example0054

Here's the machine's output:

```
index: 0 terminated
index: 2 terminated
```

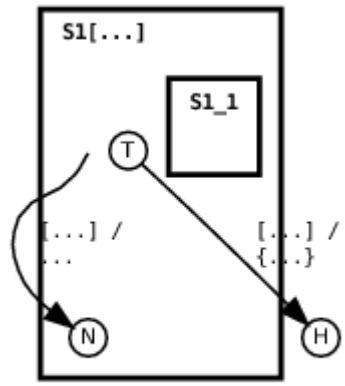
In this example, the parent machine first creates 4 machine array elements (see Section 6.4 for an explanation of the graphical depiction of this transition). Then the machine's second transition fires twice: only for array elements that are in their terminal states and whose index value modulo 2 is equal to 0.

3.11.2.3 Index Gettors

Whereas Section 3.11.2.1 shows how to obtain an array element index to be utilized by the ECharts runtime, this section deals with how to specify an array element index utilized by the ECharts runtime. The following machine provides examples of both source and target index gettors.

```
package examples;

public machine Example0053 {

    <* private final int bound = 3 *>
    <* private int i = bound *>
    <* private int j = 0 *>

    initial state S1[bound]
        entry System.out.println("created target index: " + i)
```

```
        : { initial state S1_1; }

    transition S1 - [ i > 0 ] /
        i--
    -> S1[i].NEW;

    transition S1[j].TERMINAL -
        [ i == 0 && j < bound ] / {
        System.out.println("destroyed source index: " + j);
        j++;
    }
    -> DEEP_HISTORY;
}
```



Example0053

The machine's output is:

```
created target index: 2
created target index: 1
created target index: 0
destroyed source index: 0
destroyed source index: 1
destroyed source index: 2
```

Unlike the machine in `Example0052`, the machine in this example specifies the index for each newly created array element. The target index gettor `S1[i].NEW` informs the ECharts runtime to use the value assigned to the machine's variable `i` as the index of the newly created array element (see Section 6.4 for an explanation of the graphical depiction of this transition).

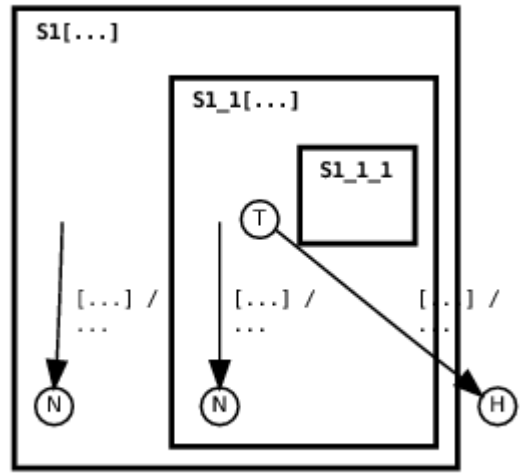
Similarly, the source index gettor `S1[j].TERMINAL` declares that the transition source state reference should be satisfied only when the array element whose index is specified by variable `i` is in a terminal state.

A precaution one must take when utilizing index gettors is to ensure that the specified array element actually exists. This means that the programmer must keep track of which array elements are free and which aren't. Like most arrays, the value of a machine array index ranges from 0 to 1 minus the array's specified bound. A free element becomes occupied when an array element is created (via a `NEW` pseudostate reference). An occupied element becomes free when the array element is destroyed. For details regarding machine creation and destruction see Section 4.7.

3.11.2.4 Nested Element References

Index settors and gettors may be used to reference nested machine array elements as shown in the following example:

```
package examples;

public machine Example0055 {

    <* final private int bound = 2 *>

    initial state S1[bound]: {
        initial state S1_1[bound]: {
            initial state S1_1_1;
        }
        <* private int created = 0 *>
         transition S1_1 - [created < bound]  / created++
        -> S1_1[created-1].NEW;
    }
    <* private int created = 0 *>
    transition S1 - [created < bound] / created++
         -> S1[created-1].NEW;

    <* int i = -1 *>

     transition [ S1[1].S1_1[?i].TERMINAL ] -
         [ created == bound ] /
         System.out.println("terminating S1[1].S1_1[" + i + "]")
     -> DEEP_HISTORY;
}
```



Example0055

The output from this machine is:

```
terminating S1[1].S1_1[0]
terminating S1[1].S1_1[1]
```

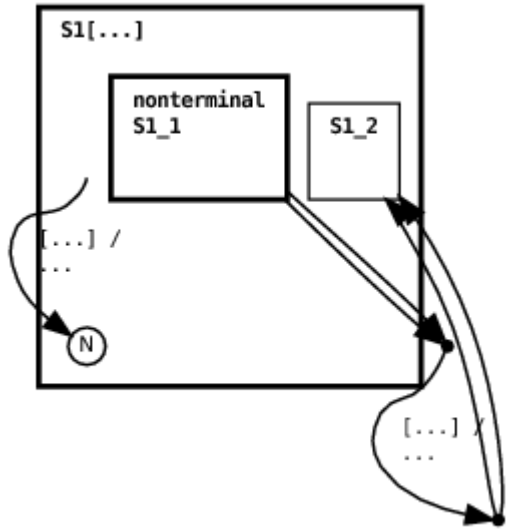
In this example the parent machine creates two machine array elements in state `S1`, and each of these elements creates two elements of their own in state `S1.S1_1` (see Section 6.4 for an explanation of the graphical depiction of this transition). The second parent transition specifies a source index gettor `S1[1]` and a source index settor `S1_1[?i]`. As a result, the second parent transition fires twice: once when `S1[1].S1_1[0]` reaches its terminal state, and once when `S1[1].S1_1[1]` reaches its terminal state.

3.11.2.5 Peer Element References

Index settors and gettors may be used to simultaneously reference peer machine array elements as shown in the following example:

```
package examples;

public machine Example0056 {

    <* final private int bound = 4 *>

    initial state S1[bound]: {
```

```
        initial nonterminal state S1_1;
        state S1_2;
    }
    <* private int created = 0 *>
    transition S1 - [created < bound] / created++
    -> S1[created-1].NEW;

    <* int i, j = -1 *>

     transition [ S1[?i].S1_1, S1[?j].S1_1 ] -
         [ created == bound && i != j ] /
         System.out.println("terminating S1[" + i +
         "] and S1[" + j + "]")
     -> [ S1[i].S1_2, S1[j].S1_2 ] ;
}
```



Example0056

One possible output from this machine is:

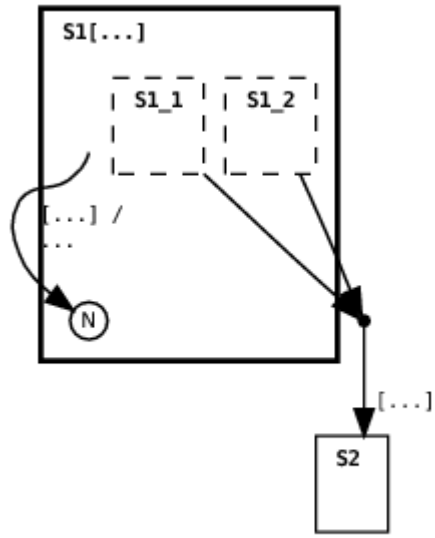
```
terminating S1[0] and S1[1]
terminating S1[2] and S1[3]
```

In the example, the parent machine creates four machine array elements in `S1` (see Section 6.4 for an explanation of the graphical depiction of this transition). The second parent transition specifies two source index settors `S1[?i].S1_1` and `S1[?j].S1_1`. The transition guard further constrains `i`

`!= j`. This means that the source state is satisfied when two distinct peer array elements are found to be in state `S1_1` (The reason for including the `nonterminal` state modifier for `S1_1` will become clear when we discuss machine destruction in Section 4.7.). The transition's target state reference specifies two target index gettors `S1[i].S1_2` and `S1[j].S1_2`. This means that if the transition fires with source index settor values `i = 0` and `j = 1`, then peer array elements `S1[0]` and `S1[1]` will both transition to state `S1_2`.

This example also shows that, in general, individual source state references may be satisfied by different machine array elements. In order to constrain source state references to refer to the same array element use the approach shown in the following example:

```
package examples;

public machine Example0058 {

    <* final private int bound = 2 *>

    initial state S1[bound]: concurrent  {
        state S1_1;
        state S1_2;
    }
    state S2;

    <* private int created = 0 *>
    transition S1 - [created < bound] / created++
    -> S1[created-1].NEW;

    <* int i, j = -1 *>

     transition [ S1[?i].S1_1, S1[?j].S1_2 ] -
     [ created == bound && i == j ] -> S2 ;
}
```



Example0058

See Section 6.4 for an explanation of the graphical depiction of the first transitions in this machine.

3.11.3 Related References

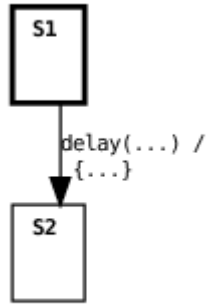
In Section 4.5.1 we discuss how message transitions are chosen to fire across machine array elements. In Section 4.8.1, we discuss how messageless transitions are chosen to fire across machine array elements. In Section 3.13 we discuss how an ancestor machine can access data and methods in machine array elements. In Section 4.8 we discuss data sharing amongst machine array elements.

3.12 Timed Transitions

Timed transitions are used for triggering timed events. A timed transition's timer is *activated* (starts ticking if it wasn't already ticking) when the state referenced by the transition's source becomes the current state. Here's a simple example:

```
package examples;
```

```
public machine Example0026 {
    initial state S1;
    state S2;
    transition S1 - delay(1000) / {
        System.out.println("duration: " + duration);
        System.out.println("activated: " + activationTime);
        System.out.println("expired: " + expiryTime);
    } -> S2;
}
```

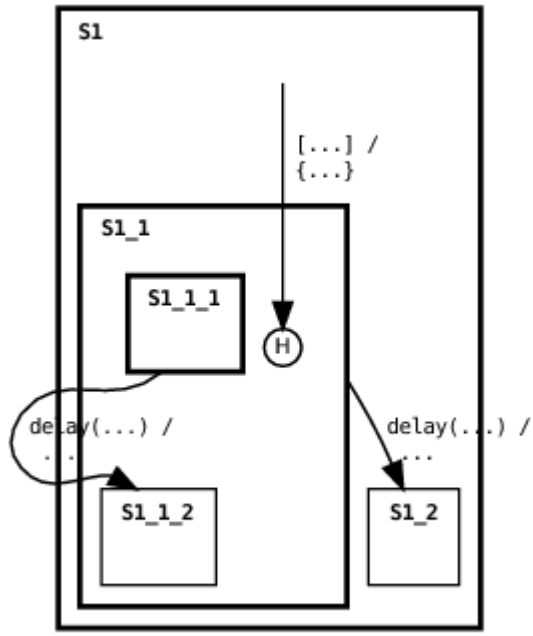


Example0026

All this machine does is pause in state `S1` for 1000 ms prior to transitioning to state `S2`. The transition's action prints out the values of three distinguished variables available during timed transition actions: `duration`, `activationTime`, and `expiryTime`. `duration` is the duration in ms associated with the transition, `activationTime` is the timestamp in ms at which the transition's timer became activated, and `expiryTime` is the timestamp in ms at which the transition's timer expired. Note that the expiry time is not necessarily equal to the time that the transition fires since other transitions may fire in the interval between the transition timer's expiry and the transition firing. For further discussion concerning the relative priority of ports and transitions, see Sections 4.4 and 4.5.

Beyond being simply activated, a timed transition's timer will be *reactivated* (activated with its counter reset) if the state referenced by the transition's source is explicitly referenced by the previously firing transition's target. On the other hand, a timed transition's timer will remain activated or expired if referenced with a `DEEP_HISTORY` pseudostate (the `DEEP_HISTORY` pseudostate is discussed in more detail in Section 3.10.3). Here's an example of both of these concepts:

```
package examples;

public machine Example0027 {

    <* private long duration = 1000 *>
    <* private boolean fired = false *>

    initial state S1: {
        initial state S1_1: {
            initial state S1_1_1;
            state S1_1_2;
            transition S1_1_1 - delay(duration) /
                System.out.println("deep duration: " + duration)
            -> S1_1_2;
        }
        state S1_2;
        transition S1_1 - delay(duration) /
            System.out.println("shallow duration: " + duration)
        -> S1_2;
    }
    transition S1 - [ !fired ] / {
        duration = 1500;
        fired = true;
    } -> S1.S1_1.DEEP_HISTORY;
}
```



In this example, the initial state of the machine activates the timers for the two timed transitions. Both timers are activated with the same duration value of 1000 ms. However, the first transition to fire is the messageless transition which changes the duration value. Because the transition's target state explicitly references state `S1.S1_1`, the timer for the (shallow) transition whose source state is `S1.S1_1` is reactivated with the new duration value (see Section 6.4 for an explanation of the graphical depiction of this transition). But the timer for the (deep) transition whose source state is `S1.S1_1.S1_1_1` remains activated with the original duration value because it is referenced with a `DEEP_HISTORY` pseudostate. Here's the machine's output:

```
deep duration: 1000
shallow duration: 1500
```

A timed transition whose duration is 0 ms will immediately be enabled. A timed transition whose duration has a negative value will never be enabled.

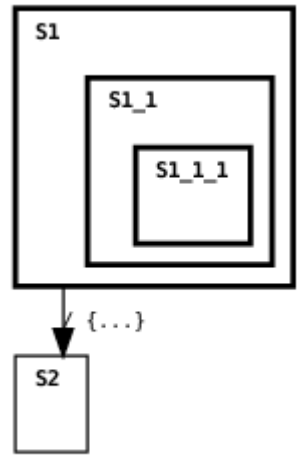
When a guard condition is specified for a timed transition, it is important that the guard's true and false conditions be handled by the transition itself using compound targets (see Section 3.4.3). This is because if the

transition's timer expires and no target is defined for the transition then an exception will be raised. See Section 4.6.1 for a detailed explanation of this behavior. The interested reader is referred to [1] for a formal description of the rules used to determine timed transition activation and reactivation.

3.13 Submachine Access

Submachine fields and methods can be accessed from transition guards, transition actions, and entry/exit actions. Every machine maintains variables for referencing its submachines. A submachine's variable name is the same as its state's name. Here's an example:

```
package examples;

public machine Example0028 {

    initial state S1: {
        <* final String field = "Hello" *>
        initial state S1_1: {
            <* final String field = "World" *>
            initial state S1_1_1;
        }
    }
    state S2;
    transition S1 - / {
        System.out.println(S1.field);
        System.out.println(S1.S1_1.field);
    } -> S2;
}
```



Example0028

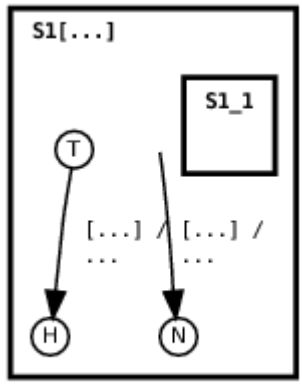
In this example, the root machine accesses the fields named `field` in submachines defined for states `S1` and `S1.S1_1`. The machine's output is:

```
Hello
World
```

Here's another example, this time illustrating machine array element access (see Section 3.11).

```
package examples;

public machine Example0044 {
    <* private final int bound = 2 *>
    <* private int created = 0 *>
    <* private final String[] messages =
        new String[] { "Hello", "World!" } *>

    initial state S1[bound]: {
        <* private final String message =
            messages[getMachineIndex()] *>
        initial state S1_1;
    }
    transition S1 - [ created < bound ] /
        created++
    -> S1.NEW;
    <* private int index = -1 *>
```

```
    transition S1[?index].TERMINAL - [ created == bound ] /
        System.out.println("terminated: " +
            S1[index].message)
    -> S1.DEEP_HISTORY;
}
```



Example0044

Here's the machine's output:

```
terminated: Hello
terminated: World!
```

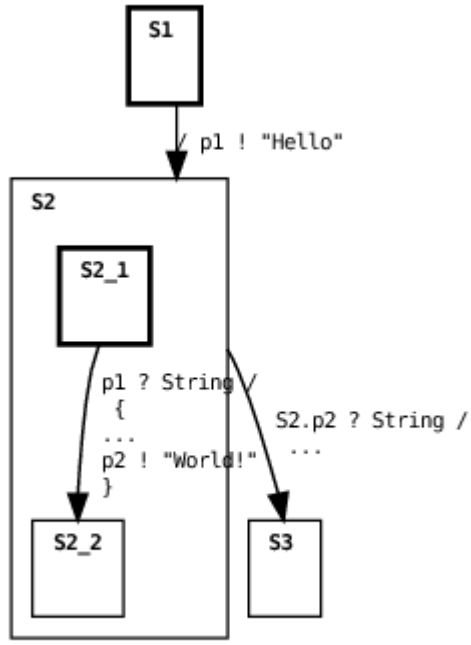
In this example, two machine array elements are initially created. The machine array elements consist only of a single state that plays the role of both the machine's initial state and its terminal state (see Section 3.10.2). Following their creation (see Section 6.4 for an explanation of the graphical depiction of this transition), the second transition is enabled to fire for each element. As described in Section 3.11.2.1, the notation `S1[ ?index ].TERMINAL` has the effect of setting the `index` variable to the value of the machine array element index satisfying the transition's source state reference. The resulting `index` value is used to access the `message` field of the referenced machine array element in the transition's action.

We discuss precisely when submachine variable values are set and cleared in Section 4.7. Submachine variable access is constrained by machine and state access permissions. This topic is discussed in Section 4.9. Also, there are constraints on shared data referenced by transitions. This is discussed in Section 4.8.

3.14 Internal Ports

Internal ports are intended to make communications between ancestor and descendant machines explicit. An internal port is nothing more than a first-in-first-out queue. Like external ports, messages can be sent and received on internal ports. Unlike external ports, an internal port does not need to be peered in order to support sending messages. Whenever possible, internal ports should be used in lieu of shared data (discussed in Section 4.8). The following example shows how internal ports are used.

```
package examples;

public machine Example0057 {

    <* final private InternalPort p1 =
        new InternalPort(this, "p1") *>

    initial state S1;
    state S2: {

        <* final private InternalPort p2 =
            new InternalPort(this, "p2") *>

        initial state S2_1;
        state S2_2;
        transition S2_1 - p1 ? String / {
            System.out.println(message);
            p2 ! "World!";
        } -> S2_2;
    }
    state S3;
    transition S1 - / p1 ! "Hello" -> S2;
    transition S2 - S2.p2 ? String /
        System.out.println(message)
    -> S3;
}
```



Example0057

The example machine output is:

```
Hello
World!
```

In the example, internal port `p1` is created by the parent machine and internal port `p2` is created by the submachine in state `S2`. A machine that creates an internal port is considered to "own" the port.

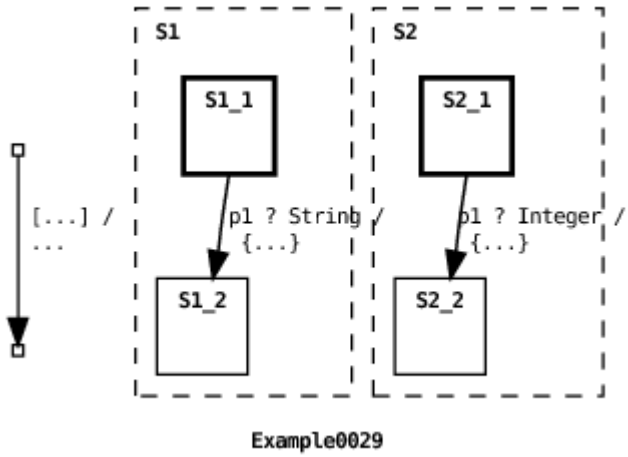
Messages are sent and received on internal ports using the same syntax used to send and receive messages on external ports (see Section 3.9). In the example, the parent machine first sends the message `"Hello"` on `p1` which is received on `p1` by the submachine. The submachine then sends the message `"World!"` on `p2` which is received on `p2` by the parent machine. The priority of an internal port is between that of a timed transition port and an external port. See Section 4.4 for more details.

3.15 Incomplete State References

A machine transition's source and target state references need not specify a state at all. When no source state is referenced then the transition's source

state is satisfied regardless of the current state of the machine. When no target state is referenced then the target state is treated as a reference to the machine's DEEP_HISTORY pseudostate (see Section 3.10.3). Here's an example:

```
package examples;

public concurrent machine Example0029 {

    <* final ExternalPort p1 *>

    public Example0029(ExternalPort p1) {
        this.p1 = p1;
    }
    <* private int messages = 0 *>
    <* private boolean end = false *>
    state S1: {
        initial state S1_1;
        state S1_2;
        transition S1_1 - p1 ? String / {
            messages++;
            System.out.println(message);
        } -> S1_2;
    }
    state S2: {
        initial state S2_1;
        state S2_2;
        transition S2_1 - p1 ? Integer / {
            messages++;
            System.out.println(message);
        } -> S2_2;
    }
    transition [] - [ !end && messages == 2 ] /
        end = true
    -> [];
}
```



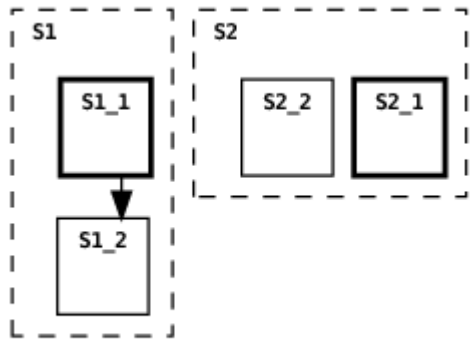

```
package examples;

import org.echarts.ExternalPort;

public class Example0029Environment {

    final public static void main(String[] argv) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
            p1.input("Hello");
            p1.input(new Integer(42));
            new Example0029(p1).run();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

In this example, the machine's environment enqueues two messages in external port `p1`'s input queue. This port is then passed to the machine as a constructor parameter. The machine maintains two concurrent submachines. One increments `messages` when a `String` instance arrives on `p1` and the other increments `messages` when an `Integer` instance arrives on `p1` (see Section 6.4 for an explanation of the graphical depiction of these transitions). When the number of messages to arrive on `p1` reaches two, the transition with no specified source or target state fires. Since this transition specifies no target state, the target is treated as a `DEEP_HISTORY` reference.

As shown in the machine diagram, the graphical notation for an empty source or target state reference is a small empty box.

When we introduced join and fork transitions in Section 3.4.2 the examples showed all of a machine's and-states being explicitly referenced. This is not necessary in general. Omitting an and-state or or-state reference in a transition's source is treated as a "don't care" reference. Omitting an and-state or or-state reference in the transition's target is treated as a `DEEP_HISTORY` reference. These points are illustrated in the following example:

```
package examples;

public concurrent machine Example0030 {
    state S1: {
        initial state S1_1;
        state S1_2;
    }
    state S2: {
        initial state S2_1;
        state S2_2;
    }
    transition S1.S1_1 --> S1.S1_2;
}
```



Example0030

In this example, the transition explicitly references and-state `S1` but not and-state `S2` (see Section 6.4 for an explanation of the graphical depiction of this transition). Since state `S2` is not referenced in the transition's source state, then `S2` is guaranteed to satisfy the transition's source state. Furthermore since `S2` is not referenced in the transition's target state, then the

reference to S2 is treated as a DEEP_HISTORY pseudostate. When the transition executes, state S1 changes from S1_1 to S1_2 and state S2 remains in S2_1.

3.16 Host Language Interface

In Section 1 we introduced ECharts as a *hosted* language. This means that ECharts was never intended to be a complete programming language. Rather, ECharts relies on an underlying host language to support data operations and low-level control flow. ECharts strives to make the boundary between itself and its host language as seamless as possible while also trying to be independent from its host language. Experience gathered from earlier versions of ECharts has led us to adopt a boundary where basic language expressions are directly supported by the ECharts language, but more complex host control-flow constructs and field and method declarations are contained within host language delimiters. Consistent with ECharts adoption of other Java concepts, ECharts has adopted Java-style expressions. These expressions are translated appropriately to the chosen target host language. Escaped host constructs delimited by <* *>, are stripped of their delimiters and inserted directly into the translated code.

It is easiest to describe which ECharts expressions are acceptable indirectly by enumerating those Java expressions that are *not* acceptable ECharts expressions and, therefore, must be wrapped in host language delimiters: instanceof, if, then, else, try, catch, throw, switch, case, do, while, for, synchronized, return, continue, break, assert, ?: ternary operator, method declarations, variable declarations, and inner class declarations.

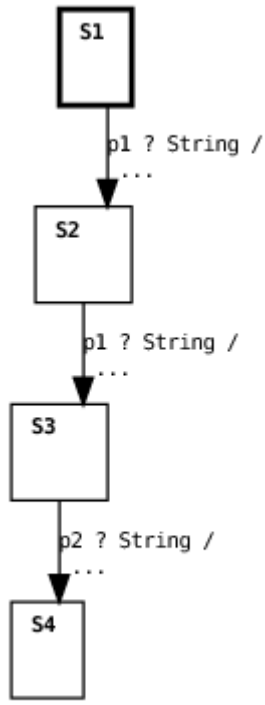
In the ECharts grammar a host language element wrapped in <* *> plays the role of a primary expression. As such it may be embedded in a larger ECharts expression. For example, a transition guard for a Java hosted machine might look like this:

```
[ <* isLongString(var1) ? var2 : var3*>.someMethod() == 42 ]
```

3.17 Machine Serialization

ECharts machines and associated classes are serializable to support their use in a high-availability environment. Serializability makes it possible to store and retrieve ECharts objects to and from a replicated data tier. Here's an example using the Javamachine runtime to save and restore a machine and a port to and from a file:

```
package examples;

public machine Example0063 {

    <* private final ExternalPort p1, p2 *>

    public Example0063(ExternalPort p1, ExternalPort p2) {
        this.p1 = p1;
        this.p2 = p2;
    }

    initial state S1;
    state S2;
    state S3;
    state S4;
    transition S1 - p1 ? String /
        System.out.println(message)
    -> S2;
    transition S2 - p1 ? String /
        System.out.println(message)
    -> S3;
    transition S3 - p2 ? String /
        System.out.println(message)
    -> S4;
}
```



Example0063

```
package examples;

import org.echarts.ExternalPort;
import org.echarts.TransitionMachine;

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;

public class Example0063Environment {

    public static final void main(String[] argv) {
        final ExternalPort p1 = new ExternalPort("p1");
        final ExternalPort p2 = new ExternalPort("p2");
        try {
            TransitionMachine machine = new Example0063(p1, p2);
            // run machine with message on port p1
            machine.run(p1, "Hello World!");
            // run machine with message on port p2 - message will be
```

```
            // enqueued
            machine.run(p2, "Bonjour le Monde!");
            ObjectOutputStream oos =
                new ObjectOutputStream(
                    new FileOutputStream("Example0063.tmp"));
            // serialize and write port p1
            oos.writeObject(p1);
            // serialize and write machine
            oos.writeObject(machine);
            oos.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
package examples;

import org.echarts.ExternalPort;
import org.echarts.TransitionMachine;

import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class Example0064Environment {

    public static final void main(String[] argv) {
        try {
            ObjectInputStream ois =
                new ObjectInputStream(
                    new FileInputStream("Example0063.tmp"));
            // read and deserialize port p1
            ExternalPort p1 = (ExternalPort) ois.readObject();
            // read and deserialize machine
            TransitionMachine machine =
                (TransitionMachine) ois.readObject();
            ois.close();
            // run machine with message on port p1 - message
            // previously enqueued on p2 will be dequeued
            machine.run(p1, "Hola Mundo!");
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }
}
```

The example machine simply waits for a message to arrive on port `p1` in states `S1` and `S2`, then it waits for a message to arrive on port `p2`. When the machine is executed with the `Example0063Environment`, the environment first runs the machine using non-blocking execution (see Section 3.6) providing a message for port `p1`. This causes the machine to transition from state `S1` to `S2`. Then the environment runs the machine providing a message for port `p2` but because no transitions in state `S2` reference `p2`, then the message is enqueued on the `p2`'s input queue and the machine remains in state `S2`. Finally, the environment serializes and writes port `p1` and the machine to the file `Example0063.tmp`. The output that results from running `Example0063Environment` is:

```
Hello World!
```

Next `Example0064Environment` is executed which reads and deserializes port `p1` and the machine from the file `Example0063.temp`. Then the environment runs the machine with a message for port `p1`. This first causes the machine to transition to state `S3`. Then, because there is a message enqueued for port `p2` and there is a transition defined for `p2` in state `S3`, the machine transitions to state `S4`. The output that results is:

```
Hola Mundo!
Bonjour le Monde!
```

See Section 5.4 for information on serialized machines with delay transitions.

Chapter 4

The Runtime Model

To this point we've examined ECharts' language features. What we've intentionally ignored so far are the fundamental semantics underlying the language. This section answers some of the questions that you may have been wondering about such as: When more than one transition can fire, which one is chosen? Can data be shared between peer machines? When are machines created and destroyed?

4.1 Machine Execution

As described in Sections 3.5 and 3.6, there are two ways to execute an ECharts machine: (1) a machine's execution blocks until a message arrives from the environment and (2) a machine method is invoked with a port and a message as arguments and, therefore, does not need to block. We describe the first mode of execution in more detail here. The non-blocking mode of execution is more complex so we omit its description for the sake of brevity.

Blocking execution of an ECharts machine proceeds as follows:

```
Fire a maximal sequence of maximum priority enabled
messageless transitions.
 while active message transitions exist in the machine's
current state:
 do
     while no messages enqueued for active ports:
     do
        Block execution until message arrives.
     end while
    Get a message from a maximum priority active port's
    input queue.
```

```
    Fire highest priority enabled message transition for the
    message and port.
    Fire maximal sequence of maximum priority enabled
    messageless transitions.
end while
```

To understand the pseudocode above you should understand the distinction between active and enabled transitions. A transition is an *active transition* if its source state reference is satisfied by the machine's current state. A transition is an *enabled transition* if (1) it is an active transition and (2) the transition's guard conditions are satisfied. A firing transition must be an enabled transition. A port is an *active port* if it is specified by an active message transition. Each machine execution cycle, a message is removed from a currently active port. More discussion related to message dequeuing is found in Section 4.6. Transition re-evaluation is discussed in Section 4.2. The concept of transition priority is discussed in Section 4.5. The concept of port priority is discussed in Section 4.4. Other issues related to execution blocking are discussed in Section 4.3. More information concerning enabled transitions can be found in Section 4.8.1.

4.2 Transition Evaluation

As described in Section 4.1, a transition is enabled when its source state references are satisfied and its guard conditions are satisfied. For reasons of efficiency the ECharts runtime does not re-evaluate all of a machine's transitions for satisfiability after a transition fires. Instead it re-evaluates a transition's source state reference and guard if (1) the machine in which the transition is declared was explicitly referenced by the previously firing transition's target or (2) the machine is an ancestor of the machine in which the previous transition fired. This re-evaluation rule affects the ability to correctly share data variables amongst machines as discussed in Section 4.8.

4.3 Transition Scheduling

The essence of ECharts machine execution described in Section 4.1 is that the ECharts runtime attempts to fire as many transitions as possible in response to receiving a message from the environment. Only when there are no more transitions to execute is the next message from the environment considered. This approach to scheduling CPU resources is not uncommon for real-time systems. It is known as "run-until-blocked" or "run-to-quiescence" scheduling.

The sequence of one message transition followed by zero or more messageless transitions is referred to as a *transition sequence*. Note that the transitions in a transition sequence may belong to different submachines, including a machine's current or-state submachine or any of its and-state submachines. However, there are constraints that dictate which transitions are considered for firing in a message sequence. This topic is addressed in Sections 4.2, 4.4 and 4.5.

A transition sequence is uninterruptible in the sense that once it is initiated for a machine no other messages from the environment will be acted upon until the sequence completes. Furthermore, the scheduling granularity is at the level of the transition. That is to say, a firing transition is guaranteed to run to completion without interruption.

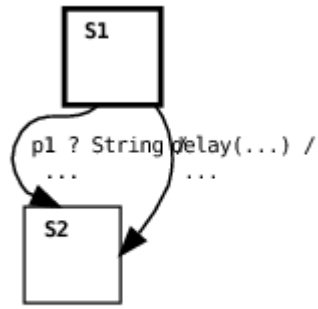
There are a number of advantages afforded by this scheduling approach. The biggest one is that there is no need for explicit concurrency control to enforce data integrity constraints. For example, a machine's transition action that updates a data value need not take precautions to protect the data value from simultaneous update by other transitions belonging to the same machine. However, it is important to realize the extent of the protection afforded by scheduling applies only to actions associated with the machine itself. For example, if the aforementioned data value is accessed by other threads (other than the machine's thread) then the usual precautions should be taken, for example using Java's `synchronized` construct. Another advantage of this scheduling policy is that scheduling granularity is effectively controlled by programmer. The less time actions take to execute, the finer the scheduling granularity.

However, this scheduling approach also has potential disadvantages that go hand in hand with with its advantages: starvation and blocking. When a transition action executes for a long time, it effectively starves other transitions from executing. For this reason, it is important that the execution time of individual actions be as kept short as possible. Furthermore, when a machine's action makes a external blocking call no other transitions in that machine can fire until that blocking call returns. For example, if a state's entry action makes a blocking call to retrieve a value from an external database, then no machine transitions can fire until the blocking call returns. In some programming contexts extended external blocking does not not matter. However, when it does matter, then efforts should be taken to convert the (synchronous) blocking call to an (asynchronous) non-blocking call that signals its completion by sending a message from the environment to a machine port.

4.4 Port Priorities

In message receive operations, the class of ports implicitly associated with timed transitions have higher priority than internal ports which, in turn, have higher priority than external ports. For example, this means that if a timed transition has expired and is ready to fire, and messages are waiting in the input queues of internal ports and external ports of message transitions that are ready to fire, then the ECharts runtime will fire the timed transition first. Here's a simple example:

```
package examples;

public machine Example0031 {
    <* final ExternalPort p1 *>
    public Example0031(ExternalPort p1) {
        this.p1 = p1;
    }
    initial state S1;
    state S2;
    transition S1 - p1 ? String /
        System.out.println("message transition fired")
    -> S2;
    transition S1 - delay(0) /
        System.out.println("timed transition fired")
    -> S2;
}
```



Example0031

```
package examples;
```

```
import org.echarts.ExternalPort;

public class Example0031Environment {

    public static final void main(String argv[]) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
            p1.input("Hello World!");
            new Example0031(p1).run();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

In this example, the timed transition expires immediately. However, the message transition is also enabled because a message was placed in the transition port's input queue by the machine's environment. Although both transitions are enabled, the timed transition is guaranteed to fire since its implicit port has higher priority than the message transition's port.

Transition priority rules are discussed in Section 4.5.

4.5 Transition Priorities

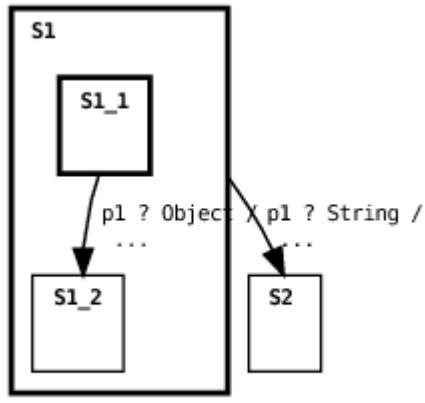
An important advantage of ECharts relative to other Statecharts dialects is that ECharts provides programmers with a number of ways to control which transition will fire when more than one transition is enabled in a machine state. Section 4.4 discusses one such way based on relative port priorities. In addition, ECharts provides three rules to resolve relative transition priority. The three priority rules are applied in the order we present them here. If one rule does not resolve the priority between comparable transitions, then the next rule is applied.

4.5.1 Message Class Rule

The first priority rule applies only to message transitions. If more than one message transition is enabled for a message from a given port, then the transition specifying the most specific message class will be chosen to fire. Here's an example:

```
package examples;
```

```
public machine Example0032 {
    <* final ExternalPort p1 *>
    public Example0032(ExternalPort p1) {
        this.p1 = p1;
    }
    initial state S1: {
        initial state S1_1;
        state S1_2;
        transition S1_1 - p1 ? Object /
            System.out.println("Got Object: " + message)
        -> S1_2;
    }
    state S2;
    transition S1 - p1 ? String /
        System.out.println("Got String: " + message)
    -> S2;
}
```



Example0032

```
package examples;

import org.echarts.ExternalPort;

public class Example0032Environment {

    public static final void main(String argv[]) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
```

```
            p1.input("Hello World!");
            new Example0032(p1).run();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

Both transitions in the machine are enabled because the message input to the port is an instance of both the `Object` and `String` classes. However, because the `String` class specified in the root machine transition is a subclass of the `Object` class specified in state `S1`'s submachine, then the `String` transition fires.

This priority rule is most often used as a way to override default behavior. One transition, intended to handle default behavior, specifies a general message class, and others are defined with more specific message classes in order to override the default transition when necessary.

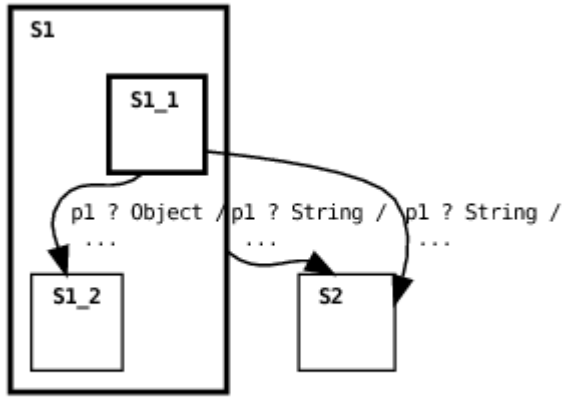
In the context of an and-state or mixed-state machine, it is possible for a port to be referenced by transitions in more than one concurrent submachine. In this case, this priority rule is applied across all submachines guaranteeing that the transition chosen to fire will be the highest priority transition across all submachines. You should realize that this can lead to poor performance when there are many concurrent submachines since potentially all submachines must be inspected to locate the highest priority transition. In contrast, when an active port is shared amongst submachine elements of a machine array (see Section 3.11) then an element with an enabled message transition is chosen non-deterministically. In this case, there is no attempt to choose the highest priority message transition across all elements since this can lead to poor performance.

4.5.2 Source Coverage Rule

The second priority rule applies to all transitions for which the first rule fails to resolve priorities. The enabled transition with the highest priority is the transition with the most specific source state reference. This rule is a generalization of what is typically the only transition priority rule supported by other Statecharts dialects. Here's an example:

```
package examples;

public machine Example0033 {
    <* final ExternalPort p1 *>
```

```
    public Example0033(ExternalPort p1) {
        this.p1 = p1;
    }
    initial state S1: {
        initial state S1_1;
        state S1_2;
        transition S1_1 - p1 ? Object /
            System.out.println("Source S1_1: " + message)
        -> S1_2;
    }
    state S2;
    transition S1 - p1 ? String /
        System.out.println("Source S1: " + message)
    -> S2;
    transition S1.S1_1 - p1 ? String /
        System.out.println("Source S1.S1_1: " + message)
    -> S2;
}
```



Example0033

```
package examples;

import org.echarts.ExternalPort;

public class Example0033Environment {

    public static final void main(String argv[]) {
        try {
```

```
            final ExternalPort p1 = new ExternalPort("p1");
            p1.input("Hello World!");
            new Example0033(p1).run();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

And here's the machine's output:

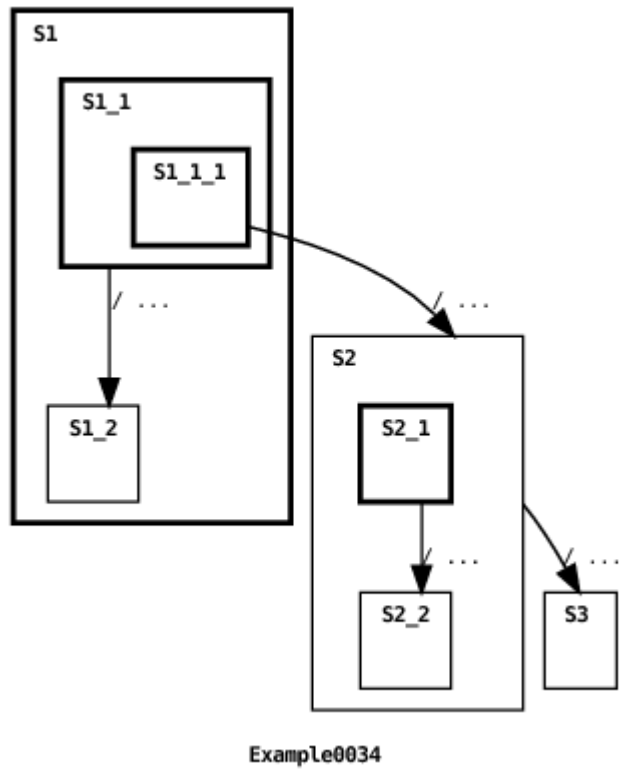
```
Source S1.S1_1: Hello World!
```

In this case all message transitions in the machine are initially enabled. But the two transitions in the root machine have higher priority than the transition in state `S1`'s submachine because of the first priority rule discussed in Section 4.5.1. However, since the two root machine transitions have equal priority according to the first rule, then the second rule is applied. Since source state `S1.S1_1` referenced by the second transition is more specific than the source state `S1` referenced by the first transition, then the second transition has higher priority than the first transition under the second priority rule, so the second transition fires.

As mentioned earlier, the second rule applies not just to message transitions but also to messageless transitions. Here are two more examples.

```
package examples;

public machine Example0034 {
    initial state S1: {
        initial state S1_1: {
            initial state S1_1_1;
        }
        state S1_2;
        transition S1_1 - /
            System.out.println("Source S1_1")
        -> S1_2;
    }
    state S2: {
        initial state S2_1;
        state S2_2;
        transition S2_1 -/
            System.out.println("Source S2.S2_1")
        -> S2_2;
    }
```

```
    state S3;
    transition S1.S1_1.S1_1_1 - /
        System.out.println("Source S1.S1_1.S1_1_1")
    -> S2;
    transition S2 - /
        System.out.println("Source S2")
    -> S3
}
```



Example0034

Here is the machine's output:

```
Source S1.S1_1.S1_1_1
Source S2.S2_1
Source S2
```

In the initial state, the root machine transition referencing source state `S1.S1_1.S1_1_1` and the submachine transition referencing source state `S1_1` are enabled. However, because the transition in the root machine references a more specific source state than the submachine transition then the root

machine transition fires. Two more transitions are enabled once the machine transitions to state `S2`: the root machine transition referencing source state `S2` and the submachine transition referencing source state `S2_1`. In this case, the submachine transition is guaranteed to fire first because its source state is more specific than the root machine transition's. Finally, the parent machine fires since it remains enabled after the submachine transition fires,

The second priority rule is useful for two purposes. One purpose is to override the default transition behavior defined for a machine source state. To do this the programmer defines additional transitions defined for particular source substates. This use of the rule is shown in the last two examples. Another purpose is to allow submachine transitions to fire before more general ancestor machine transitions fire. This supports the view of a submachine invocation being akin to a procedure call in a traditional language. This use of the rule is shown in the last part of the second example above.

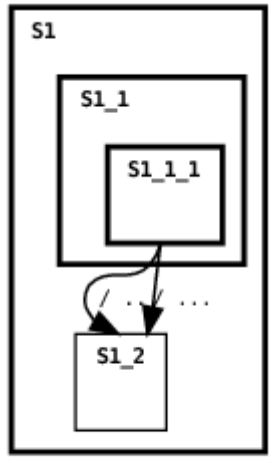
When the `TERMINAL` pseudostate (see Section 3.10.2) is referenced by a transition's source, then it is considered to be less specific than a reference to an actual machine state. The second priority rule also generalizes to join transitions (discussed in Section 3.4.2). In this case, each join transition branch reference must be less specific in order for the entire join to be considered less specific. The notion of source state *specificity* is formally captured by *term coverage* which we will not cover in this document in the interests of brevity. The curious reader is referred to [3] for more information.

4.5.3 Transition Depth Rule

This last rule is applied when the previous two rules fail to resolve the relative priority of two comparable transitions. Like the source coverage rule discussed in Section 4.5.2, this rule applies to all types of transitions: timed, messageless and message transitions. When the source states references of two transitions are identical, this rule gives highest priority to the transition defined in a higher-level machine (or, if you prefer, a shallower depth machine). Here's an example:

```
package examples;

public machine Example0035 {
    initial state S1: {
        initial state S1_1: {
            initial state S1_1_1;
```

```
        }
        state S1_2;
        transition S1_1.S1_1_1 - /
            System.out.println("Source S1_1.S1_1_1")
        -> S1_2;
    }
    transition S1.S1_1.S1_1_1 - /
        System.out.println("Source S1.S1_1.S1_1_1")
    -> S1.S1_2;
}
```



Example0035

Here's the machine's output:

```
Source S1.S1_1.S1_1_1
```

In this example both transitions refer to the same source state, namely state `S1.S1_1.S1_1_1`. In this case, the application of the transition depth rule guarantees that the root machine's transition fires (see Section 6.4 for an explanation of the graphical depiction of this transition), since the root machine depth is shallower (depth 0) than the depth of the submachine defined for state `S1` (depth 1).

The motivation for using this rule is to override submachine behavior. If a programmer would like to alter the behavior of a particular submachine's transition, then a transition in an ancestor machine can be defined to reference the same source state and, optionally, the same target states as the submachine transition. This priority rule ensures that the ancestor transition will fire instead of the submachine's transition.

4.6 Message Dequeuing

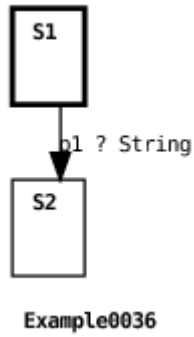
In Sections 3.5 and 3.8 devoted to receiving and sending messages, we discussed how an external port maintains an input queue into which messages from a machine's environment are enqueued. In Section 4.4 we discussed how dequeuing messages from timed transition ports has higher priority than dequeuing messages from other message transition ports. In this section we discuss two other issues related to dequeuing messages. The ECharts approach to message dequeuing attempts to balance two conflicting desires: (1) to never unintentionally lose messages from the environment and (2) to not overly burden the programmer. These two issues are addressed, in turn, in the next two sections.

4.6.1 Explicit Message Consumption

The ECharts runtime model guarantees that no messages from the environment are unintentionally lost. The justification for this is simply that in most reactive programming domains, it is unacceptable to unknowingly lose a message. This aspect of ECharts distinguishes it from other Statecharts dialects that typically allow a message to be lost if it is not explicitly handled by a machine.

Here's how ECharts ensures that messages are not lost: if p is an active port (see Section 4.1), then the ECharts runtime may dequeue a message from port p. Furthermore, if a message is dequeued from port p, then the machine *must* fire some message transition for that message, otherwise an exception is raised. Here's an example:

```
package examples;

public machine Example0036 {
    <* final private ExternalPort p1 *>
    public Example0036(ExternalPort p1) {
        this.p1 = p1;
    }
    initial state S1;
    state S2;
    transition S1 - p1 ? String -> S2;
}
```



Example0036

```
package examples;

import org.echarts.ExternalPort;

public class Example0036Environment {
    final static public void main(String[] argv) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
            p1.input(new Integer(42));
            new Example0036(p1).run();
        } catch (Exception e) { System.out.println(e); }
    }
}
```

Here's the machine's output:

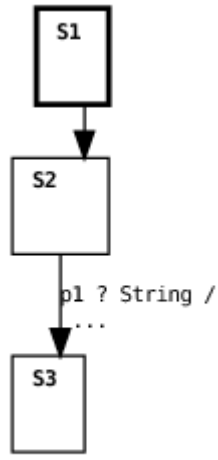
```
org.echarts.MachineException: No transition defined for
message class java.lang.Integer with string representation 42
on port PORT_NAME=p1, PORT_TYPE=EXTERNAL_PORT,
PORT_ID=192.168.1.102:5420:109c366d015:-8000
from state :examples.Example0036 S1 (S2)
```

In this example, the machine's environment enqueues an `Integer` instance on port `p1`'s input queue. However, the machine only defines a message transition for a `String` instance on `p1`, so the ECharts runtime raises an exception that is caught by the environment. If there had been an additional message transition for `p1` that specified the `Integer` class or the `Object` (super) class then no exception would have been raised.

4.6.2 Implicit Message Deferral

Having seen how unhandled messages are treated in the previous section, you may be asking yourself if it is necessary to add message transitions to every machine state for every message class anticipated on every port. The answer to this question is an emphatic: No. This would be an unreasonable burden on the programmer. Furthermore, it would obfuscate a machine's logic. Instead, ECharts provides programmers with a facility to implicitly defer dequeuing messages from a port until the programmer explicitly declares a machine's readiness to accept a message from the port. The way this is accomplished is very simple: if a port p is not active (see Section 4.1), then the ECharts runtime will not dequeue a message from p. Only when the machine arrives in a state where p is active may a message be dequeued from p. If a message is dequeued from p then the machine is obliged to fire a transition for that message. Here's an example:

```
package examples;

public machine Example0037 {
    <* final ExternalPort p1 *>
    public Example0037(ExternalPort p1) {
        this.p1 = p1;
    }
    initial state S1;
    state S2;
    state S3;
    transition S1 --> S2;
    transition S2 - p1 ? String /
        System.out.println(message)
    -> S3;
}
```



Example0037

```
package examples;

import org.echarts.ExternalPort;

public class Example0037Environment {

    final static public void main(String[] argv) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
            p1.input("Hello World");
            new Example0037(p1).run();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

The example shows the machine environment enqueuing a message on port `p1`. However, the initial state of the machine does not include transitions specifying `p1` so the message remains enqueued on the port until the machine reaches state `S2` where there is a transition specifying `p1`. Since this transition specifies a `String` message then it fires and the machine completes its execution.

4.7 Machine Lifecycle

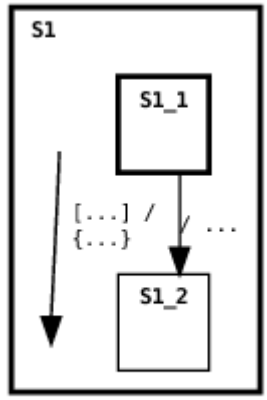
In the interests of conserving memory, ECharts machines are created only when they are needed and they are destroyed when they are no longer needed. This section explains in more detail what is meant by "needed" and "no longer needed."

4.7.1 Machine Creation

The root machine is created by the machine's environment, as we have seen in many of the previous examples. However, submachines can be created in one of three ways: (1) If a submachine does not already exist then it will be created when it becomes part of a machine's current state. (2) If a submachine already exists, then it will be *re-created* if its parent state is explicitly referenced as a target of a firing transition, and no sub-states of the parent state are referenced. (3) Finally, a machine array element can be created using the `NEW` pseudostate as discussed in Section 3.10.4.

Rule (1) ensures that a machine always exists when you expect it to. Rule (2) can be useful in order to explicitly reset submachine state. Here's an example of both rules:

```
package examples;

public machine Example0038 {

    initial state S1: {
        <* private int field = 42 *>

        initial state S1_1;
        state S1_2;
        transition S1_1 - /
            System.out.println(field)
        -> S1_2;
    }
    <* private boolean fired = false *>

    transition S1 - [ !fired ] / {
        fired = true;
        S1.field = 0;
    } -> S1;
}
```



Example0038

Here's the machine's output:

```
42
42
```

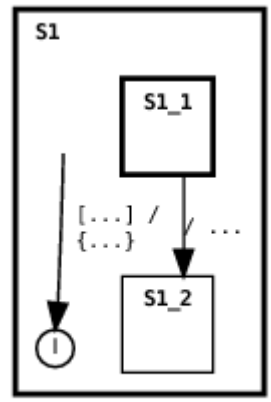
In `Example0038`, the submachine defined for initial state `S1` is automatically created when the root machine is initially created. This is because `S1` is the initial current state of the machine. The submachine transition fires printing out the initial value of `field`: 42. Then the parent transition fires (see Section 6.4 for an explanation of the graphical depiction of this transition). Since the transition target explicitly references `S1` and no substates of `S1`, then `S1`'s submachine is re-created. The evidence of this is that, although the parent transition changes the value of the submachine's field to 0, the value of submachine's field printed when the submachine transition fires again is its initial value: 42.

The above example shows how rule (1) ensures that or-state submachines are guaranteed to exist when they are expected to exist. This rule generalizes to and-state and mixed-state machines in the expected way. Since every and-state is part of the machine's current state, then all and-state submachines are guaranteed to exist during the lifetime of the parent machine.

To highlight the behavior of rule (2) consider the following example:

```
package examples;

public machine Example0039 {
```

```
    initial state S1: {
        <* private int field = 42 *>

        initial state S1_1;
        state S1_2;
        transition S1_1 - /
            System.out.println(field)
        -> S1_2;
    }
    <* private boolean fired = false *>

    transition S1 - [ !fired ] / {
        fired = true;
        S1.field = 0;
    } -> S1.DEFAULT_INITIAL;
}
```



Example0039

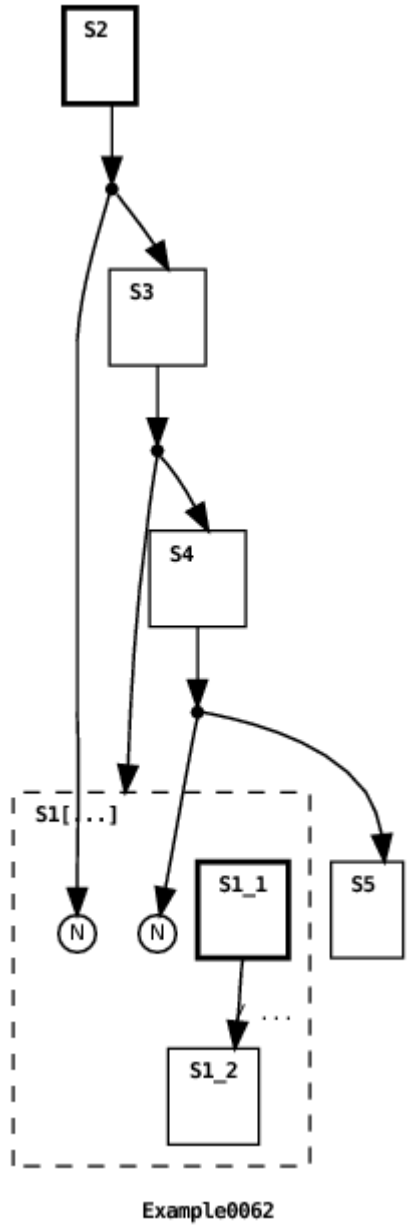
This example is identical to the previous example except that the root transition target references `S1`'s `DEFAULT_INITIAL` pseudostate, not simply `S1` as before (see Section 6.4 for an explanation of the graphical depiction of this transition). In this case, `S1`'s submachine is not re-created since the transition references a sub(pseudo)state of `S1`. This is evident from the program's output:

```
42
0
```

For practical purposes, rule (2) has been modified somewhat when ap-

plied to a machine array state. Instead of re-creating all the existing machine array elements, the array is cleared so that no elements exist in the array. Here's an example:

```
package examples;

public machine Example0062 {

    concurrent state S1[2]: {
        initial state S1_1;
        state S1_2;
        transition S1_1 - /
            System.out.println("Hello World") -> S1_2;
    }
    initial state S2;
    state S3;
    state S4;
    state S5;
    transition S2 --> [ S3, S1.NEW ];
    transition S3 --> [ S4, S1 ];
    transition S4 --> [ S5, S1.NEW ];
}
```



The machine output is:

```
Hello World
Hello World
```

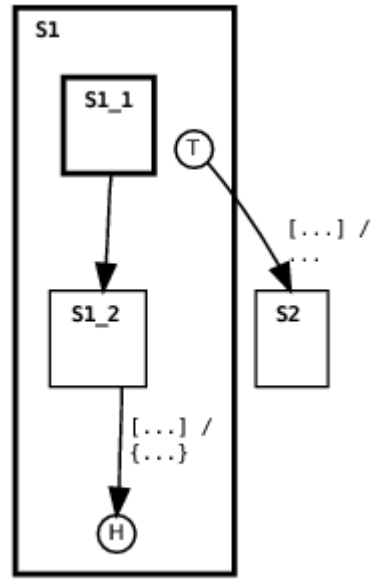
In this mixed-state machine example, the first parent machine transition

creates an array element in the and-state machine array `S1`. The next parent transition clears the machine array. The last parent transition creates another new array element. Note that `S1`'s array size is `1` so if the machine array hadn't been cleared by the second parent transition, the third parent transition would have resulted in a runtime exception being thrown.

4.7.2 Machine Destruction

Apart from destruction as a part of re-creation (as discussed above in Section 4.7.1), a machine can only be destroyed once it enters a terminal state (the concept of a terminal state is defined in Section 3.10.2). However, a machine is not destroyed upon entering a terminal state; it can only be destroyed by an ancestor machine. In particular, the source state of a transition in an ancestor machine must explicitly reference the descendant machine's terminal state in order for the descendant machine to be destroyed. Furthermore, the descendant machine isn't destroyed until the transition's actions have completed, thereby providing an opportunity to access the descendant machine's methods or fields immediately prior to its destruction. Here's an example:

```
package examples;

public machine Example0040 {
    initial state S1: {
        <* private int field = 42 *>
        initial state S1_1;
        state S1_2;
        transition S1_1 --> S1_2;
    }
    state S2;
    <* private boolean fired = false *>
    transition S1.S1_2 - [ !fired ] / {
        System.out.println("S1.field: " + S1.field);
        S1.field = 0;
        fired = true;
    } -> S1.DEEP_HISTORY;
    transition S1.TERMINAL - [ fired ] /
        System.out.println("S1.field: " + S1.field)
    -> S2;
}
```



Here's the machine's output:

```
S1.field: 42
S1.field: 42
```

In this example, the submachine in state `S1` transitions to a terminal state. Then the first root machine transition fires (see Section 6.4 for an explanation of the graphical depiction of this transition), printing the initial value of the submachine's `field` variable, changing its value to 0, and then looping back to the `DEEP_HISTORY` pseudostate of the submachine. Since the submachine was in an terminal state, and since the root machine transition explicitly referenced the terminal state, then the submachine will have been destroyed when the transition fired. This means that the changed value of the submachine's `field` variable will be lost when the submachine is destroyed. The transition's target is state `S1`, so `S1` becomes the machine's current state again. As described in the previous section, Section 4.7.1, this means that a new instance of the submachine in `S1` must be created. Since the newly created submachine has no prior state, the transition target `DEEP_HISTORY` pseudostate reference defaults to the submachine's initial state `S1_1`. Once again, the submachine's transition fires, moving the submachine to a terminal state. Now the second transition in the root machine fires, printing out the current value of the submachine's `field` variable. Since the

variable is defined in a newly created submachine instance then the initial field value, 42, is printed. You should also note that since the transition source references the submachine with the `TERMINAL` pseudostate, then the submachine will, once again, be destroyed. This time, however, the new current state of the machine becomes `S2`, so the submachine in state `S1` is not re-created.

The same rule for machine destruction applies to elements of a machine array (see Section 3.11). In the context of a machine array, it is important for the programmer to remember to include transitions to destroy machine array elements that have arrived in a terminal state. Neglecting to do this will eventually result in the machine array reaching its maximum capacity leaving no opportunity for new elements to be added. Furthermore, this situation may also over-utilize memory resources. Also, as discussed in Section 4.7.1, it is possible to explicitly destroy all machine array elements at once, effectively clearing the array.

The destruction of a machine can be prevented by annotating the machine's parent state with the `nonterminal` state modifier. As an example, see `Example0022` in Section 3.11. This modifier informs the ECharts interpreter to treat the parent state as a non-terminal state even though it might be a terminal state. The `nonterminal` modifier does not affect machine re-creation as described in Section 4.7.1, however.

4.8 Shared Data

This section discusses the use of shared variables between machines. While data sharing may sometimes be necessary, internal ports should be used in lieu of shared data whenever possible. See Section 3.14 for details.[1]

4.8.1 Guard Variables

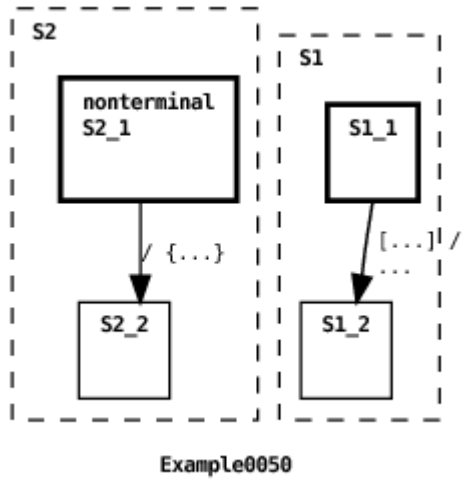
To correctly utilize shared variables in transition guards one needs to understand the transition evaluation rule described in Section 4.2. Note that the rule's two conditions imply that only ancestor and descendant machines may share variables referenced by guards; peer machines should not directly share variables referenced by guards.

To gain an intuition for the rule in the context of variable sharing here are a few examples. The first example illustrates an *incorrect* use of shared

[1]The `shared` machine modifier, present in ECharts versions less than or equal to 1.1 Beta, has been eliminated from the language due to the semantic complexity and performance overhead it entailed.

variables with guards. This will be followed by two examples showing the correct use of shared variables with guards.

```
package examples;

public concurrent machine Example0050 {

    state S1: {
        <* private boolean flag  = false *>
        initial state S1_1;
        state S1_2;
        transition S1_1 - [ flag ] /
            System.out.println("World!")
        -> S1_2;
    }
    state S2: {
        initial nonterminal state S2_1;
        state S2_2;
    }
    transition S2.S2_1 - / {
        S1.flag = true;
        System.out.println("Hello");
    }
    -> S2.S2_2;
}
```



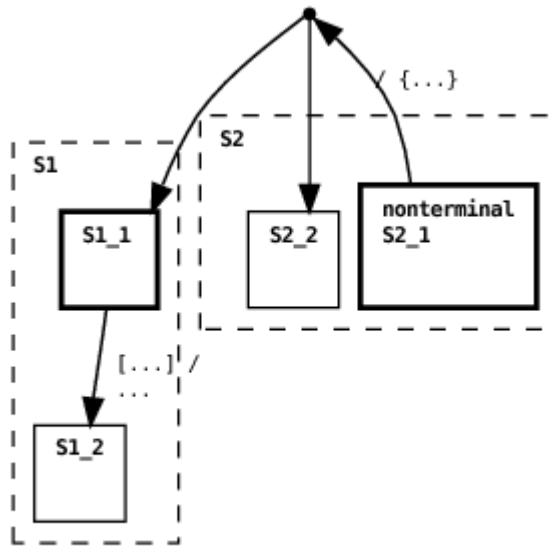
In the above example we show a failed attempt by a transition declared in the parent machine to trigger a transition in child machine `S1` whose

guard references a shared variable `flag` (see Section 6.4 for an explanation of the graphical depiction of this transition). The program produces the following output.

```
Hello
```

The parent updates the variable's value but, because the program violates part (1) of the guard evaluation rule, the child's transition does not fire. In particular, the parent transition does not explicitly reference `S1` which means that `S1`'s transition guard is not re-evaluated after `flag`'s value is updated.

```
package examples;

public concurrent machine Example0051 {

    state S1: {
        <* private boolean flag  = false *>
        initial state S1_1;
        state S1_2;
        transition S1_1 - [ flag ] /
            System.out.println("World!")
        -> S1_2;
    }
    state S2: {
        initial nonterminal state S2_1;
        state S2_2;
    }
    transition S2.S2_1 - / {
        S1.flag = true;
        System.out.println("Hello");
    }
    -> [ S1.S1_1, S2.S2_2 ];
}
```



The problem with `Example0050` is rectified in the example above. In this case, the parent machine transition explicitly references the child machine `S1` via the target state reference `S1.S1_1`. In accordance with part (1) of the guard evaluation rule, `S1`'s transition guard is re-evaluated and the transition fires. The program's output is:

```
Hello
World!
```

Another example of correctly sharing variables referenced by guards is `Example0029` in Section 3.15. In that example, the variable `messages` is declared in the parent machine and updated by transitions declared in the two concurrent child machines. The `messages` variable is also referenced by the parent machine's transition guard. Since the parent is an ancestor of the child machines then, in accordance with part (2) of the guard evaluation rule, a transition firing in a child machine (that updates the `messages` value) will result in the parent's transition guard being re-evaluated, which is the desired behavior.

4.8.2 Port Variables

A message transition specifies a port variable. Normally a port variable's value remains constant during a machine's lifetime. However, occasionally

it is desirable to change a port variable's value at runtime so the question arises: when is a transition's port value re-evaluated? The answer is: the same rule that holds for the evaluation of guard variables described in Section 4.2. The same implications also hold, namely, that only ancestor and descendant machines may update shared port variables; peer machines should not directly update one another's shared port variables.

4.9 Machine and State Access Modifiers

Access permissions can be specified for ECharts machines, machine constructors and machine states. The permissions model used by ECharts is the same as that used by Java. There are four permission classes: `private` (currently unsupported for machines), `public`, `protected` (currently unsupported), and package (the default permission class if no permission class is explicitly specified). Machine pseudostates are assigned `public` access.

The permissions for machines, machine constructors and machine states are interpreted the same way as for Java class definitions, instance constructors, and instance methods, respectively.

Machine permissions and state permissions dictate submachine access permissions (see Section 3.13) and submachine state reference permissions in multi-level transitions (see Section 3.4.1). Machine permissions and machine constructor permissions dictate permissions for specifying and creating external submachines (see Section 3.1.2).

Chapter 5

The Machine Runtime

An ECharts runtime executes ECharts machines (see Section 2.1). There are a number of configuration options and classes shared by ECharts runtimes to support runtime tasks such as monitoring and debugging. These are examined in the following sections.

5.1 Initialization

All ECharts runtime options can be configured when the runtime is initialized. There are two techniques for configuring options when the runtime is initialized: (1) using a properties file and (2) using command-line options. Some of these options can also be configured programmatically after initialization by invoking methods defined by the ECharts runtime API.

If option configuration techniques are combined then the following precedence order is obeyed: method invocation overrides command-line options, and command-line options override properties file settings.

For the *javamachine* runtime, a properties file takes the usual Java properties file format where each line in the file is of the form:

```
option: value
```

Command-line options for the *javamachine* runtime take the usual format for defining Java virtual machine system properties with the `java` command, namely:

```
-Doption=value
```

We discuss the various options in detail in the following sections.

5.2 Properties File

If the ECharts runtime finds a file named *echarts.properties* in the runtime's working directory then it will use the properties definitions found in that file. The runtime command line option `org.echarts.properties.dir` may also be used to specify a path to a directory other than the working directory in which to find the *echarts.properties* file.

5.3 Startup Messages

When an ECharts runtime is initialized, the initialization options can be printed to standard output by setting the `org.echarts.system.startupMessages` option. This option is disabled (`false`) by default but can be enabled by setting the option to `true`. This option can only be configured from a properties file or from the command-line.

5.4 Transition Timer Manager

An ECharts runtime uses a default transition timer manager to support timed transitions. However, for some application domains it is appropriate to use a domain-specific transition timer manager. For example, the ECharts interface for SIP servlet containers defines its own transition timer manager. The `org.echarts.system.transitionTimerManager.class` property specifies a fully qualified class name for the ECharts runtime to use as its transition timer manager. The specified class must implement the `org.echarts.TransitionTimerManager` interface. The default value for this property is `org.echarts.DefaultTransitionTimerManager`. The runtime loads the specified class and creates an instance of it for use by the runtime. This option can only be configured from a properties file or from the command-line.

The default transition timer manager for the Javamachine runtime is not serializable because it was not designed for operation in a highly available environment. For this reason, delay transitions of a restored serialized machine (see Section 3.17) will never fire. However, since the ECharts runtime permits overriding the default transition timer manager, it is possible to avoid this constraint. For example, the transition timer manager implementation included with the ECharts for SIP Servlets Development Kit [7] uses the highly available timer facility provided by the underlying SIP servlet container.

5.5 Monitoring and Logging

An ECharts runtime includes a simple, configurable, extensible monitoring subsystem. This subsystem receives events generated by the runtime, optionally filters out particular events, and then formats and logs the events. The runtime uses this subsystem to support debugging, however, it can be used for many other purposes such as monitoring application-specific events. It can also easily be integrated with an application's existing monitoring subsystem.

To log an event, an ECharts application must send it to a monitor (see Section 5.5.1). For example, the built-in ECharts trace debugger (see Section 5.6) sends a series of events to ECharts machine monitors during the course of a debugging session. Events are submitted to a monitor via the monitor's `putEvent()` method (see Section 5.5.3). This method checks if the event passes through the monitor's event filter (see Section 5.5.4) and, if it does, then it logs the event using the monitor's formatter (see Section 5.5.5).

5.5.1 Monitors

A runtime's monitoring subsystem is disabled by default. When monitoring is disabled, no events are sent to the monitoring subsystem. To enable it, set the `org.echarts.system.monitoring` property to `true`, or invoke the `setSystemMonitoring()` method. In addition to enabling monitoring for the runtime, monitoring must be enabled for the individual machines from which you want to receive events. The simplest way to do this is to set the `org.echarts.machine.monitoring` property to `true` or invoke the `setDefaultMachineMonitoring()` method. This value is set to `false` by default. The value of this property is used as the default monitoring setting for newly created machines. This default value can be overridden for a particular machine by calling the `setMachineMonitoring()` method. The default behavior is that a child machine inherits the monitoring setting of its parent, so if you wish to use different monitoring settings in a parent and its child, you must use the `setMachineMonitoring()` method.

Each machine managed by an ECharts runtime has a monitor. Machines may share a monitor, or they may use different monitors. The `org.echarts.machine.monitor.class` property can be used to specify a fully qualified classname of a monitor. The specified class must implement the `org.echarts.monitor.MachineMonitor` interface. The runtime will load the class and create an instance of it to use as the default monitor for newly created machines. The default value for this property is

`org.echarts.monitor.PrintStreamMonitor` (discussed in Sections 5.5.5 and 5.6). This default value can be overridden for a particular machine by calling the `setMachineMonitor()` method. The default behavior is that a child machine inherits the monitor of its parent, so if you wish to use different monitors between parent and child, you must use the `setMachineMonitor()` method.

For a discussion of monitor formatters and logging, see Section 5.5.5.

5.5.2 Events

All monitor events are instances of the ECharts `MachineMonitorEvent` class. When an event instance is created it is automatically timestamped. An event also possesses a properties list of string key/value pairs. When an event is logged, as described in 5.5.5, then the event's timestamp and properties list are printed.

Apart from the pre-defined trace debugger event classes, there is one additional pre-defined event class, the `InfoEvent` class. This event class can be used for a number of purposes by the programmer, for example, logging custom debug messages or exception stack traces. However, it is also possible to create custom event classes by simply subclassing the root `MachineMonitorEvent` class, or the `InfoEvent` class itself. `Example0046` in Section 5.6.1 provides an example of using the `InfoEvent` class.

5.5.3 putEvent()

To submit an event to a monitor, the monitor's `putEvent()` method must be invoked. A `putEvent()` method is also defined for each machine instance. The programmer may choose which of these `putEvent()` methods to invoke. Invoking a machine's `putEvent()` method causes the input event to be augmented with four properties prior to calling the machine monitor's `putEvent()` method. These additional properties are intended to associate the event with the machine producing the event. `Example0046` in Section 5.6.1 provides an example of using a machine's `putEvent()` method.

The `MACHINE_ID` and `ROOT_MACHINE_ID` properties specify the unique ID assigned to the machine instance producing the event and its root machine, respectively. The `MACHINE_STATE_PATH` property specifies the state path from the root machine to the machine producing the event. The `SEQUENCE_ID` property specifies the unique ID assigned to the transition sequence that the event is a member of (see Section 4.3 for a description of transition sequences). A sequence ID is actually the unique ID assigned to the port associated with the message transition that initiates the sequence.

If it is not desirable to augment events with these properties, then a machine monitor's `putEvent()` method should be invoked directly rather than indirectly invoking it via the machine's `putEvent()` method.

5.5.4 Event Filters

When a machine submits an event to its monitor via its `putEvent()` method, the monitor first passes the event through its event filter. If the event passes through the filter the event is logged, otherwise it is discarded. Defining an event filter for a monitor provides flexible, fine-grained control of a monitor's logging levels.

An event filter is an instance of the `MachineMonitorEventFilter` class. A filter string is specified in the class constructor. A filter string specifies passing/discarding an event based on its (sub-)class or on its property values. A null or empty filter string passes all events. For example, here is an event string that passes `InfoEvent` instances whose `MESSAGE` property value is `Hello`, and discards `InfoEvent` instances whose `MESSAGE` property is `World`:

```
+org.echarts.monitor.InfoEvent?MESSAGE=(Hello) |
-org.echarts.monitor.InfoEvent?MESSAGE=(World)
```

The filter string syntax supports more complex filters including the specification of regular expressions for property values and conjuncts of event classes. A complete description can be found in the `MachineMonitorEventFilter` class documentation.

The default event filter for the ECharts runtime's default monitor (see Section 5.5.1) can be set during initialization with the `org.echarts.machine.monitor.filterString` property, or at runtime with the `setDefaultMachineMonitorFilter()` method. The default value for this property is the empty string, which passes all events.

5.5.5 Formatters and Logging

Event logging refers to what a monitor does with an event once it passes through the monitor's event filter. Typically, the event is logged to a file. Event formatting refers to how an event is portrayed in the event log. The logging task is performed by the monitor itself while the formatting task is performed by a monitor's formatter. An ECharts runtime comes with four general purpose monitor classes: (1) a `NullMonitor` that effectively discards all events sent to it, (2) a `PrintStreamMonitor` that logs formatted events to a specified print stream, for example, a file or standard output, (3) a `LoggerMonitor` that logs formatted events to a specified

`java.util.logging.Logger` instance (which supports log file rotation, among other things) and (4) a `RawMonitor` that logs unformatted events in their raw, binary format to a specified file. An ECharts runtime also comes with two formatter classes: (1) a `PrettyPrintFormatter` which formats events as easy-to-read strings, and (2) a `TextFormatter` which simply stringifies events in a machine-readable format. We will describe the output of the `PrettyPrintFormatter` in more detail in Section 5.6 devoted to the trace debugger.

A `PrintStreamMonitor` using a `PrettyPrintMonitor` is the default value for the ECharts runtime default monitor. Section 5.5.1 describes how the default monitor value can be set.

Programmers are free to extend any of the monitor or formatter classes for their own purposes in order to, for example, integrate ECharts runtime monitoring with an existing logging subsystem, or to customize runtime monitoring to suit their application needs.

5.6 Debugging

An ECharts runtime supports trace debugging. When enabled, trace debugging generates a sequence of monitor events reflecting machine execution steps, namely, transition events and machine creation or destruction events.

5.6.1 Output Format

The actual format of the events depends on the event formatter utilized by a machine's monitor (see Section 5.5.5). In the examples shown in this section we will use the `PrettyPrintFormatter`. Here's the complete trace debugger output produced when running the "Hello World" example from the beginning of this document (Section 2.1).

```
time: 2006.03.31 23:12:22:141 EST (1143864742141)
event: MessagelessTransitionEvent
root: a42792:10a53a6d0e6:-8000
machine: a42792:10a53a6d0e6:-8000
state path: :(Example0001)
sequence: initializing
transition: [S1] --> [S2]
local state:
    ...
    S2
    ...
```

The `time` and `event` fields specify the time the event was created and the event's class, respectively. Transition events are not created until immediately *after* the associated transition has fired. Since trace debug events are submitted to a machine's `putEvent()` method, they include the four properties discussed in Section 5.5.3. The `PrettyPrintFormatter` formats these properties as the `root`, `machine`, `state path`, and `sequence` fields.

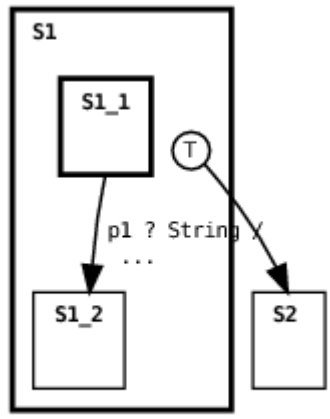
In this example, the `root` and `machine` fields specify the same machine since the root machine fired the transition. Since there are no submachines in this example, the `state path` simply specifies the root machine. Since the firing transition belongs to the initial sequence of messageless transitions that fire when a root machine is initialized by the runtime the ID is shown as `initializing`.

The next two fields are common to message transition and messageless transition events. The `transition` field specifies the source and target state references of the messageless transition that fired. Had the transition been a message transition, then the transition's specified message class and a string representation of the message satisfying the transition are also displayed. In the case that a transition defines a compound target (see Section 3.4.3), then only the chosen target state reference is displayed. Finally, the `local state` field specifies the updated state of the machine in which the transition fired. Only states explicitly referenced by the transition's target state are displayed. In the event that the affected states contain submachines, then the submachine states are also shown.

Now here's a more complex example. First we show the machine and its environment. Then we show the trace debugger output resulting from running the machine's environment.

```
package examples;

public machine Example0046 {

    <* final private ExternalPort p1 *>

    public Example0046(ExternalPort p1) {
        this.p1 = p1;
    }
    initial state S1: {
        initial state S1_1;
        state S1_2;
        transition S1_1 - p1 ? String /
            putEvent(new InfoEvent(message))
        -> S1_2;
```

```
    }
    state S2;
    transition S1.TERMINAL --> S2;
}
```



Example0046

```
package examples;

import org.echarts.ExternalPort;

public class Example0046Environment {

    public static final void main(String[] argv) {
        try {
            final ExternalPort p1 = new ExternalPort("p1");
            p1.input("Hello World!");
            new Example0046(p1).run();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

```
time: 2006.03.31 23:48:38:154 EST (1143866918154)
event: MachineLifecycleEvent
root: ac2f9c:10a53c803f8:-7fff
machine: ac2f9c:10a53c803f8:-7fff
state path: :(Example0046)
```

```
sequence: initializing
created: S1:(inner machine)
submachine: ac2f9c:10a53c803f8:-7ffe
################################################################
time: 2006.03.31 23:48:38:221 EST (1143866918221)
event: InfoEvent
root: ac2f9c:10a53c803f8:-7fff
machine: ac2f9c:10a53c803f8:-7ffe
state path: :(Example0046).S1
sequence: ac2f9c:10a53c803f8:-8000
message: Hello World!
################################################################
time: 2006.03.31 23:48:38:224 EST (1143866918224)
event: MessageTransitionEvent
root: ac2f9c:10a53c803f8:-7fff
machine: ac2f9c:10a53c803f8:-7ffe
state path: :(Example0046).S1
sequence: ac2f9c:10a53c803f8:-8000
transition: [S1_1] - p1 ? String (Hello World!) -> [S1_2]
local state:
{
    ...
    S1_2
    ...
}
################################################################
time: 2006.03.31 23:48:38:249 EST (1143866918249)
event: MachineLifecycleEvent
root: ac2f9c:10a53c803f8:-7fff
machine: ac2f9c:10a53c803f8:-7fff
state path: :(Example0046)
sequence: ac2f9c:10a53c803f8:-8000
destroyed: S1:(inner machine)
submachine: ac2f9c:10a53c803f8:-7ffe
################################################################
time: 2006.03.31 23:48:38:281 EST (1143866918281)
event: MessagelessTransitionEvent
root: ac2f9c:10a53c803f8:-7fff
machine: ac2f9c:10a53c803f8:-7fff
state path: :(Example0046)
sequence: ac2f9c:10a53c803f8:-8000
transition: [S1.TERMINAL] --> [S2]
local state:
{
    ...
    S2
```

```
    ...
}
############################################################
```

The trace debugger output from this example shows two `MachineLifecyleEvents`. These events are associated with submachine creation and destruction. The `created` and `destroyed` fields indicate the parent state of the submachine being created or destroyed, respectively. It also shows the submachine's class if the submachine is an external machine, otherwise it shows `inner machine` as illustrated above. The `submachine` field specifies the submachine instance's unique ID. The trace also shows how an `InfoEvent` is displayed (see Section 5.5.2). Using `InfoEvent`'s is preferable to using a host language print statement because the event is logged and correlated in time with other trace debugger events. Furthermore, if the event was submitted to a machine's `putEvent()` method as described in Section 5.5.3 then the event will be automatically augmented with a number of additional properties.

The examples above show the default output format produced by the `PrettyPrintFormatter`. The `PrettyPrintFormatter` also has methods to control the display of debugger events including how machine class names and machine states are portrayed. See the `PrettyPrintFormatter` class API for details.

5.6.2 Message Properties

A message transition event includes a string representation of the message that triggered the transition by default. For the *javamachine* runtime, the string is the result of calling the message object's `toString()` method. However, it is possible to include more structured information about the message in the form of a properties list of string key/value pairs. Not only is the properties list maintained in a structured format in a monitor log, the individual keys and values can also be referenced in a event filter in a straightforward fashion (see Section 5.5.4). For the *javamachine* runtime, a message class should implement the `org.echarts.monitor.Message` interface to associate a properties list with a message. When a message transition event is created for a message implementing this interface, the message properties are added to the event's properties. For an example, see the `org.echarts.TransitionTimeoutMessage` class. An instance of this class is sent by the ECharts runtime to a timer port when a timed transition expires. The message specifies three properties: `DURATION`, `ACTIVATION_TIME`, and `EXPIRY_TIME`.

5.6.3 Options

The simplest way to enable trace debugging for a runtime consists of setting a single option: `org.echarts.debugging`. When this option is set to `true` then the appropriate debugging and monitoring options are set so that debugger events are sent to monitors for all machine instances.

If fine-grained control over debugging is desired then there are a number of other options available similar to those for monitors (see Section 5.5.1). First, monitoring must be enabled for a runtime and for the individual machines you wish to monitor in order for debug events to be received by the monitoring subsystem. Next, it is also necessary to enable debugging for the runtime and the individual machines you wish to monitor.

A runtime's trace debugger is disabled by default. To enable it, set the `org.echarts.system.debugging` property to `true`, or invoke the `setSystemDebugging()` method. In addition to enabling debugging for the runtime, debugging must be enabled for the individual machines you want monitor. The simplest way to do this is to set the `org.echarts.machine.debugging` property to `true` or invoke the `setDefaultMachineDebugging()` method. This value is set to `false` by default. The value of this property is used as the default debugging setting for newly created machines. This default value can be overridden for a particular machine by calling the `setMachineDebugging()` method. The default behavior is that a child machine inherits the debugging setting of its parent, so if you wish to use different debugging settings in a parent and its child, you must use the `setMachineDebugging()` method.

If the system or machine debugging properties are disabled then debug events are not created and sent to monitors, regardless of a runtime's monitor settings. Nonetheless, with monitoring enabled and debugging disabled it is possible to log non-debug events, such as `InfoEvent` instances (see Section 5.5.2).

5.6.4 Global State Output

The `local state` field described in Section 5.6.1 provides a local view of machine state. Sometimes it is also desirable to obtain a global view, that is, a snapshot of a machine's current or-states and and-states from the root machine downwards, recursively including the current states of all submachines. For even moderately complex machines, this snapshot can be hard to understand given the amount of information contained in it. Nevertheless it can sometimes be useful to periodically examine global machine state. For this reason, runtimes support printing global machine state im-

mediately following the execution of a transition sequence (see Section 4.3). Here's an example of the output produced by the `PrettyPrintFormatter` when this option is enabled.

```
time: 2006.03.31 23:54:34:764 EST (1143867274764)
event: MachineStateEvent
root: ac2f9c:10a53cd74c2:-7fff
machine: ac2f9c:10a53cd74c2:-7fff
sequence: initializing
global state:
:Example0046{
    S1{
        S1_1
    }
}
```

This event corresponds to the global state of the `Example0046` machine (discussed in Section 5.6.1) immediately after the initialization sequence that causes the creation of state `S1`'s submachine. In this example, the current machine or-state is state `S1` and its submachine's current or-state is `S1_1`.

Root machines and their monitors are responsible for producing global state events so be sure that you minimally enable debugging for the root machine of the machine you wish to collect global state events from. Furthermore, you must explicitly inform the runtime that you wish to collect global state events since this capability is disabled by default. To enable this capability for all newly created machine instances, set the `org.echarts.machine.debugging.globalStateOutput` to `true`, or invoke the `setDefaultMachineDebuggingGlobalStateOutput()` method. To enable or disable this capability for a particular (root) machine instance, use the `setMachineDebuggingGlobalStateOutput()` method.

5.7 Options Summary

Property	
Default Value	Section
`org.echarts.properties.dir`	
`.`	5.2
`org.echarts.system.startupMessages`	
`false`	5.3
`org.echarts.system.transitionTimerManager.class`	
`org.echarts.DefaultTransitionTimerManager`	5.4
`org.echarts.system.monitoring`	
`false`	5.5.1
`org.echarts.machine.monitoring`	
`false`	5.5.1
`org.echarts.machine.monitor.class`	
`org.echarts.monitor.PrintStreamMonitor`	5.5.1
`org.echarts.machine.monitor.filterString`	
`""`	5.5.1
`org.echarts.debugging`	
`false`	5.6.3
`org.echarts.system.debugging`	
`false`	5.6.3
`org.echarts.machine.debugging`	
`false`	5.6.3
`org.echarts.machine.debugging.globalStateOutput`	
`false`	5.6.4

Chapter 6

Generating Diagrams

ECharts provides the `ech2dot` translator program for generating ECharts machine diagrams suitable for inclusion in reports, web pages, or presentations. It also provides translators for generating browsable code documentation. The documentation translators are discussed in Section 7.

All these translators share in common their dependence on the open source Graphviz `dot` program [4] for generating graphical depictions of ECharts machines. The `dot` program takes as input a text file (with a *.dot* extension) specifying the constituent nodes and arcs of a directed graph. Its output is a graphical depiction of the graph in one of a number of possible graphical output formats e.g. PostScript, SVG, or PNG. The `.dot` files themselves are generated from ECharts machine files by the `ech2dot` translator.

Since there are so many ways to generate ECharts diagrams we've included a variety of examples here: Section 6.1 discusses generating diagrams as separate pages and Section 6.2 discusses generating diagrams suitable for embedding in other documents. In Section 6.3 we explain how one can customize a generated diagram if the default output format is unsatisfactory. Finally, in Section 6.4, we discuss how bugs in the current version of `dot` can affect a diagram's layout.

Depending on the desired diagram format, some or all of the following open source utilities (not included in the ECharts SDK) are used to assist with diagram generation: `dot`[1] for laying out and rendering the diagram in a given output format, `gs`[2] for converting PostScript to PostScript or PDF (Portable Document Format), `eps2pdf`[3] for converting encapsulated

[1] Distributed as part of the open source Graphviz package available from `http://graphviz.org`

[2] GNU Ghostscript is available from `http://gnu.org`

[3] Available from `http://www.ctan.org/tex-archive/support/eps2pdf`

PostScript to PDF, and `inkscape`[4] for editing SVG diagrams. All examples here assume the working directory to be *runtime/java/src/examples* directory of the ECharts SDK. For information on configuring your platform to run ECharts commands like `ech2dot` see Appendix A.

6.1 Page Diagrams

Here are a number of examples of generating a diagram as a printable page. In these examples we use the following Unix conventions: the command line prompt is a percent character `%`, long commands are split over multiple lines using the backslash character `\` at the end of a line, and the command line continuation prompt for a command split over multiple lines is the 'greater than' character `>`.

Generate a 8.5 x 11 inch portrait layout diagram in PostScript format with 0.5 inch margins. We post-process the dot output with `gs` (ghostscript) in order to add explicit page size information.

```
% ech2dot --echartspath .. Example0001.ech
% dot -Tps -Gsize="7.5,10" -o Example0001.temp \
> Example0001.dot
% gs -q -dNOPAUSE -dBATCH -sPAPERSIZE=letter \
> -sDEVICE=pswrite -sOutputFile=Example0001.ps \
> Example0001.temp
```

Generate a 8.5 x 11 inch (US letter) landscape layout diagram in PostScript format with 0.5 inch margins.

```
% ech2dot --echartspath .. Example0001.ech
% dot -Tps -Gorientation=landscape -Gsize="10,7.5" \
> -o Example0001.temp Example0001.dot
% gs -q -dNOPAUSE -dBATCH -sPAPERSIZE=letter \
> -sDEVICE=pswrite -sOutputFile=Example0001.ps \
> Example0001.temp
```

Generate a 8.5 x 11 inch landscape layout diagram in PDF (Portable Document Format) with 0.5 inch margins. We post-process the dot output with `gs` in order to translate to PDF and to add explicit page size information.

```
% ech2dot --echartspath .. Example0001.ech
% dot -Tps -Gorientation=landscape -Gsize="10,7.5" \
```

[4] Available from `http://inkscape.org`

```
> -o Example0001.temp Example0001.dot
% gs -q -dNOPAUSE -dBATCH -sPAPERSIZE=letter \
> -sDEVICE=pdfwrite -sOutputFile=Example0001.pdf \
> Example0001.temp
```

Generate a 210 x 297 mm (A4) landscape layout diagram in PDF with 0.5 inch margins.

```
% ech2dot --echartspath .. Example0001.ech
% dot -Tps2 -Gsize="10.69,7.27" -Gorientation=landscape \
> -o Example0001.temp Example0001.dot
% gs -q -dNOPAUSE -dBATCH -sPAPERSIZE=a4 \
> -sDEVICE=pdfwrite -sOutputFile=Example0001.pdf \
> Example0001.temp
```

Generate a 36 x 22 inch portrait layout diagram in PostScript format with increased font size and 1 inch margins. This non-standard poster size page is suitable for printing on a 36 inch wide plotter. We post-process the dot output with `gs` in order to add explicit page size information.

```
% ech2dot --echartspath .. Example0001.ech
% dot -Tps2 -Gsize="34,20" -Efontsize=18 \
> -Elabelfontsize=12 -Gfontsize=18 \
> -o Example0001.temp Example0001.dot
% gs -q -dNOPAUSE -dBATCH -sDEVICE=pswrite \
> -dDEVICEWIDTHPOINTS=2592 -dDEVICEHEIGHTPOINTS=1584 \
> -sOutputFile=Example0001.ps Example0001.temp
```

6.2 Embedded Diagrams

Here are a few examples of how to generate a diagram for embedding in a document page, for example in a web page or a report. In these examples we use the following Unix conventions: the command line prompt is a percent character `%`, long commands are split over multiple lines using the backslash character `\` at the end of a line, and the command line continuation prompt for a command split over multiple lines is the 'greater than' character `>`.

Generate a 6 x 4 inch diagram in PNG (Portable Network Graphics) with increased font size.

```
% ech2dot --echartspath .. Example0001.ech
% dot -Tpng -Efontsize=12 -Elabelfontsize=8 \
> -Gfontsize=12 -Gsize="6,4" -o Example0001.png \
> Example0001.dot
```

Generate a 6 x 4 inch diagram in SVG (Scalable Vector Graphics) format.

```
% ech2dot --echartspath .. Example0001.ech
% dot -Tsvg -Gsize="6,4" -o Example0001.svg \
> Example0001.dot
```

Generate a 6 x 4 inch diagram in PostScript format. We post-process the `dot` output with `gs` in order to translate to EPS (encapsulated PostScript).

```
% ech2dot --echartspath .. Example0001.ech
% dot -Tps2 -Gsize="6,4" -o Example0001.temp \
> Example0001.dot
% gs -q -dNOPAUSE -dBATCH -sDEVICE=epswrite \
> -sOutputFile=Example0001.ps Example0001.temp
```

Generate a 6 x 4 inch diagram in PDF. We post-process the `gs` output with `eps2pdf` in order to translate to embeddable PDF.

```
% ech2dot --echartspath .. Example0001.ech
% dot -Tps2 -Gsize="6,4" -o Example0001.temp1 Example0001.dot
% gs -q -dNOPAUSE -dBATCH -sDEVICE=epswrite \
> -sOutputFile=Example0001.temp2 Example0001.temp1
% eps2pdf --outfile=Example0001.pdf Example0001.temp2
```

6.3 Customizing the Layout

There are four ways to customize diagrams generated with `ech2dot`:

1. editing the *.dot* file output by `ech2dot` prior to submitting it to `dot`;
2. generating an SVG format file with `dot` and then editing the diagram with `inkscape`.
3. customizing the `dot` program's output by overriding layout parameters on the `dot` command line;
4. customizing the `ech2dot` program's output by specifying custom formatters to override the default formatters;

The first option requires an understanding of the Graphviz `dot` language which we do not discuss here. Instead the interested user is referred to [5] for more information. Generating the SVG file for the second option is discussed in Section 6.1. We discuss the last two options in more detail in the following sections.

6.3.1 Overriding `dot` Layout

The `ech2dot` translator generates Graphviz *.dot* files specifying particular values for layout parameters. Of these, the most relevant for customizing the layout's appearance are its size and orientation, its label font name and sizes, its layout ratio and its printed caption.

`ech2dot` specifies the default size for a diagram to be 8 x 10.5 inches with portrait layout. The diagram size and orientation can be overridden via the `dot` command line as shown in the examples above.

The translator also specifies 8 point font size for edge (transition) and subgraph (state) labels, and the graph (machine) caption, and a 6 point font size for edge head and tail labels. The CourierNew bold font is used for all labels. As shown in the examples above, it is possible to override these font sizes on the command line. It is also possible to override the font name in a similar fashion. For example, to use Helvetica font instead of CourierNew bold:

```
% dot -Tps2 -Efontname="Helvetica" \
> -Elabelfontanme="Helvetica" -Gfontname="Helvetica" \
> -o Example0001.ps Example0001.dot
```

Note that both a machine's layout and its states' layout are dictated via `dot`'s `-G` parameter so it is not possible to specify different font names or font sizes for a machine's caption and its states.

The `ech2dot` translator also specifies the "fill" layout ratio. For this parameter value, dot attempts to fill as much of the area specified by the diagram's size as possible. It does this by expanding the size of subgraphs (states) and arcs (transitions). For simple machines with large specified size this can result in oversized nodes. To generate a diagram that satisfies the diagram's specified size without increasing node size you can try setting the ratio to null. A diagram generated this way will have natural node sizes but is not guaranteed to fill the area specified by the diagram's size. For example:

```
% dot -Tps2 -Gratio="" -o Example0001.ps Example0001.dot
```

The default caption for an ECharts machine diagram is its (unqualified) class name. It is possible to override the caption with arbitrary text via the `dot` command line as shown in the following example:

```
% dot -Tps2 -Glabel="Arbitrary Text" -o Example0001.ps \
> Example0001.dot
```

The diagram caption can also be overridden by specifying a custom `ech2dot` formatter as explained in Section 6.3.2.

There are many other layout options available for `dot`. The interested reader is referred to [5] for more information.

6.3.2 Overriding `ech2dot` Layout

When `ech2dot` translates a *.ech* file to a *.dot* file it utilizes methods defined in modules called *formatters* to obtain formatted representations of information included in the *.dot* file. The formatted information in the *.dot* file is then included in a machine diagram output by the `dot` program. For example, a formatter can specify how a machine state or transition declaration is portrayed in a machine diagram.

The `ech2dot` program permits a user to define their own formatters if the default formatters are not suitable for their purposes. There are three formatter classes that may be overridden by the user:

label formatter This formatter defines methods for formatting state and transition labels appearing on a machine diagram.

tooltip formatter This formatter defines methods for formatting tooltips that are embedded in client-side imagemap files output by `dot`.

URL formatter This formatter defines methods for formatting URLs that are embedded in client-side imagemap, PostScript, or SVG files output by `dot`.

As described in the `ech2dot` command reference in Section 8, there are three command line options available for overriding the three formatter classes: `--label-formatter`, `--tooltip-formatter`, and `--url-formatter`, respectively. The value for each parameter must be a valid formatter (Python) class name. A formatter class is normally a subclass of the parent formatter class *translator/lib/dotmachine/DotMachineFormatter.py* included in the ECharts SDK. This class provides default definitions for methods that can be overridden by formatter subclasses. To support the default behavior of the `ech2dot` translator, four formatter subclasses are included with the ECharts SDK in the *translator/lib/dotmachine* directory:

- *DotMachineNullFormatter.py* As its name implies, this formatter returns the empty string for all state and transition formatter method calls.

- *DotMachineCompleteFormatter.py* The methods of this formatter return the complete state or transition definition string as found in the original *.ech* file.

- *DotMachinePartialFormatter.py* The methods of this formatter return a partial state or transition definition string as found in the original *.ech* file. For example, state label strings only include the state name and the submachine class name, but any submachine parameters are excluded.

- *DotMachineCommentFormatter.py* The methods of this formatter return the user comments associated with a state or a transition.

Here are the default values of the three `ech2dot` formatters:

Formatter	Class Name
label	DotMachinePartialFormatter.DotMachinePartialFormatter
tooltip	DotMachineCommentFormatter.DotMachineCommentFormatter
URL	DotMachineFormatter.DotMachineFormatter

6.4 `dot` Layout Bugs

The open source directed graph layout program `dot` is used to produce the example machine diagrams included in this manual. The layouts produced by this program are normally excellent, however, the current version of `dot` (version 2.15) suffers from bugs that cause some transitions to be depicted incorrectly. In particular, there are two problems related to the depiction of a transition referencing the same source and target states i.e. looping transitions:

1. `dot` does not always correctly depict the machine level a transition is declared at;

2. `dot` does not always connect a transition to its referenced source or target states.

As an example of both layout problems consider the depiction of `Example0021` in Section 3.11. The graphical depiction of the transition in state `S1` shows no source state reference. A correct depiction would show the transition source connected to the border of state `S1`. Furthermore, the transition should be depicted as extending outside `S1`'s border since the transition is declared in the `Example0018` machine, not in the state `S1` submachine. The same problems are seen in `Example0022`, `Example0052`,

`Example0054`, `Example0053`, `Example0055`, `Example0056` and `Example0058` in Section 3.11, `Example0025` in Section 3.10.2, `Example0027` in Section 3.12, `Example0044` in Section 3.13, `Example0029` and `Example0030` in Section 3.15, `Example0035` in Section 4.5.3, `Example0038` and `Example0039` in Section 4.7.1, `Example0040` in Section 4.7.2, and `Example0050` in Section 4.8.

Chapter 7

Generating Documentation

The `ech2doc` and `ech2javadoc` translators are used to support the generation of browsable ECharts machine code documentation. The output from both translators includes interactive SVG ECharts machine diagrams. As such, both translators rely on `ech2dot` and `dot`, the programs used to generate ECharts machine diagrams described in Section 6.

7.1 Interacting with Diagrams

There are a number of ways that a user may interact with the ECharts SVG diagrams included in the generated documentation. Viewing and interacting with an SVG diagram requires that a web browser be configured with an SVG viewer plugin. See Section 7.4 for more information.

1. Highlight transitions and states by rolling the cursor over them.

2. Display user comments for a machine, its transitions and its states by resting the cursor over the respective machine element.

3. Clicking on a state with a nested submachine navigates the browser to the diagram of the submachine nested in the state.

4. Pan a diagram that may be too large to fit in the viewing window. For Internet Explorer, hold the 'alt' key while dragging the diagram. For Safari, hold the 'shift' key while dragging. For Firefox, click on the 'pan' tool icon and then hold the 'shift' key while dragging.

5. Zoom in and out of a diagram. For Internet Explorer, hold the 'control' key and click to zoom in at the mouse pointer location. Hold the 'control' key and click-and-drag to select a region to zoom into. Hold

the 'control' and 'shift' keys and click to zoom out. You can also use the zoom commands in the context menu. For Safari, use the 'View -¿ Make Text Bigger'/'View -¿ Make Text Smaller' menu options or use the keyboard shortcuts 'command +'/ 'command -'. For Firefox, select the 'zoom' tool icon and scroll your mousewheel or press the '+'/'-' keys.

7.2 `ech2doc`

The `ech2doc` translator generates a web site whose pages provide a host-language independent overview of ECharts machine code. This program's output and options are similar to those provided by Java's `javadoc` tool. The `ech2doc` translator only generates documentation for ECharts `.ech` files and not for any supplemental host language files that may exist. Furthermore, pages generated by the `ech2doc` translator only include user comments for the machine, its constructors, its states and its transitions. The pages do not include user comments for host language constructs such as field or method declarations.

Here's an example of using the `ech2doc` translator for the `examples` machine package included with the ECharts SDK. In this example we assume the working directory to be *runtime/java/src/examples* of the ECharts SDK. We also adopt the following Unix conventions: the command line prompt is a percent character `%`, long commands are split over multiple lines using the backslash character `\` at the end of a line, and the command line continuation prompt for a command split over multiple lines is the 'greater than' character `>`.

```
% ech2doc --echartspath .. \
> --target-directory /tmp/ech2doctest examples
```

Running this command creates a web site in the directory */tmp/ech2doctest* with the front page (*/tmp/ech2doctest/index.html*) shown in Figure 7.1. Clicking on the Example0001 link in the All Machines frame displays the page shown in Figure 7.2.

The `ech2doc` translator supports many of the same command line options as Java's `javadoc` tool. These options are discussed in the `ech2doc` command reference in Section 8.

7.3 `ech2javadoc`

Unlike the `ech2doc` translator discussed in Section 7.2 which is a host language independent documentation generator, the `ech2javadoc` generator

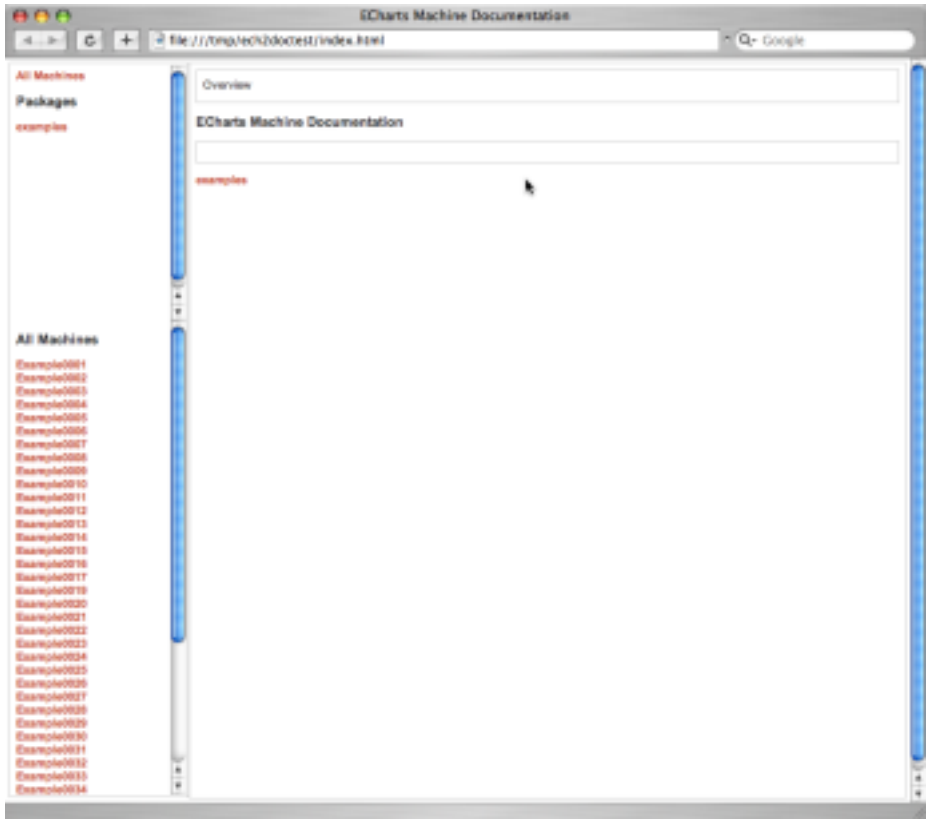


Figure 7.1: `ech2doc` front page

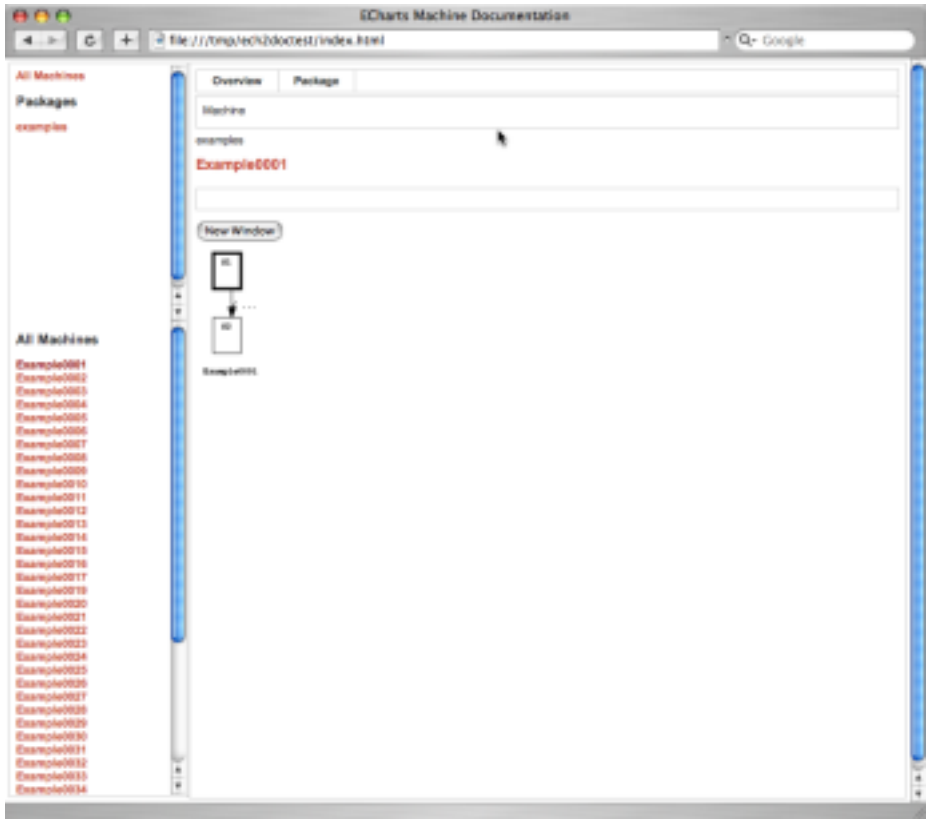


Figure 7.2: `ech2doc` Example0001 page

discussed in this section is intended for ECharts machines translated to Java using `ech2java`. The standard program for documenting Java classes and packages is `javadoc`[1]. The ECharts SDK provides two utilities that enhance `javadoc` for documenting Java source code that includes ECharts machines: (1) the `ech2javadoc` translator which generates interactive SVG diagrams for ECharts machines and (2) the `javadocpp` program which post-processes ECharts machine class documentation produced by `javadoc` in order to incorporate machine diagrams into the documentation.

Here are the commands required to generate `javadoc` documentation for the `examples` package included with the ECharts SDK. We assume the working directory to be *runtime/java/src/examples* of the ECharts SDK.

```
% ech2javadoc --echartspath .. examples
% ech2java --echartspath .. examples
% javadoc -sourcepath .. -d /tmp/ech2javadoctest examples
% javadocpp /tmp/ech2javadoctest
```

The first command (`ech2javadoc`) translates the ECharts machines in the `examples` package into SVG diagrams and places them in the *docfiles* subdirectory of the *examples* directory in which the machines are located. The `ech2javadoc` command also creates an auxiliary `.html` file for each SVG diagram and places these files in the same *docfiles* subdirectory. Running the second command (`ech2java`) is only required if ECharts machines haven't yet been translated to Java. The ECharts machines must be translated to Java in order to be referenced by `javadoc`. The third command (`javadoc`) generates documentation for the Java files in the `examples` package and places it in the */tmp/ech2javadoctest* directory. The fourth command, `javadocpp`, post-processes the documentation generated by `javadoc` to customize the documentation for ECharts machines (see Section 8 and Appendix A for information on running this command and the ECharts translator commands). In particular, `javadocpp` removes documentation for any constructors, methods or fields that may be introduced by the `ech2java` translator but are not explicitly declared in a machine by the programmer. The program also adds a button onto documentation pages for machines. As an example, Figure 7.3 shows the page generated for the `Example0001` machine. Note the button labeled Machine. Clicking on this button opens a new window displaying the machine's interactive SVG diagram as shown in Figure 7.4.

The `ech2javadoc` translator supports a number of command line options for customizing its behavior. These are enumerated in the `ech2javadoc` command reference in Section 8.

[1] Included with the Java SDK.

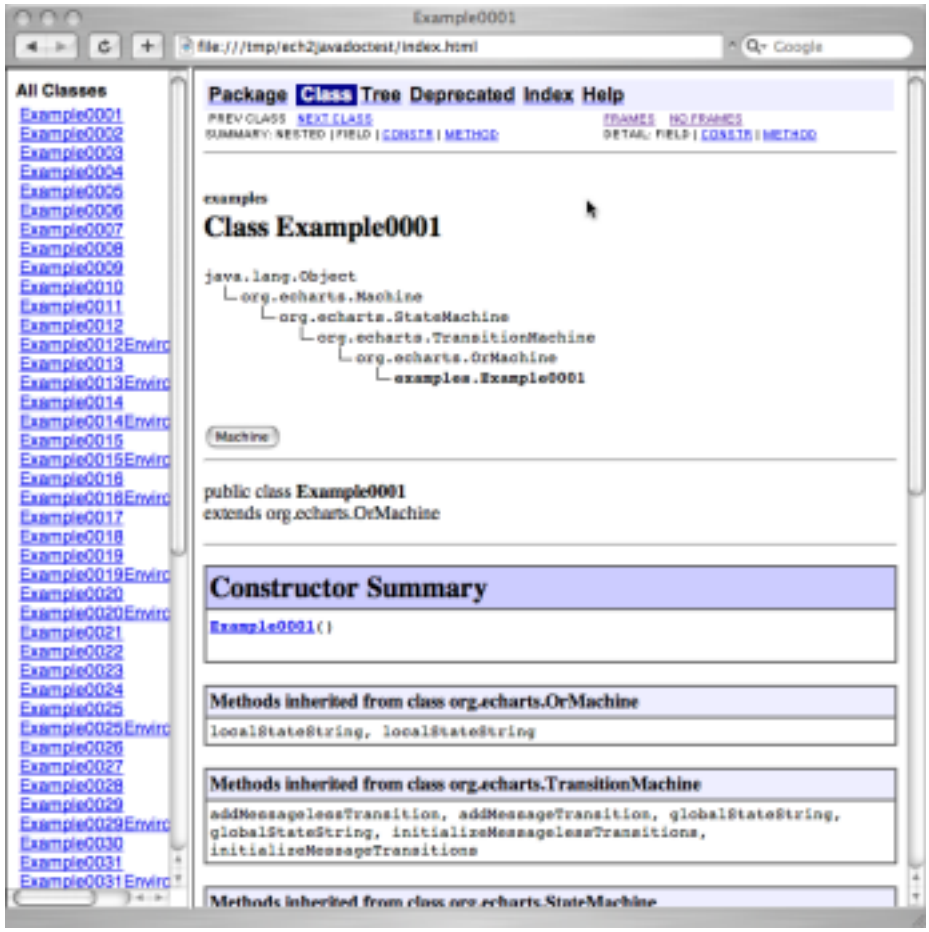


Figure 7.3: `ech2javadoc` Example0001 page

Figure 7.4: `ech2javadoc` Example0001 diagram

7.4 SVG Viewers

Viewing and interacting with SVG machine diagrams embedded in ECharts documentation web pages requires a web browser with SVG support. Both Safari version 3.x (for Mac OS X and Windows) and Firefox version 2.x (for Mac OS X, Linux and Windows) provide native SVG support. Firefox users should download the "SVG Zoom and Pan" Firefox extension[2] which provides pan/zoom support. For Microsoft Internet Explorer we recommend using Adobe's SVG viewer browser plug-in version 3.x[3].

[2]Available from `http://www.treebuilder.de/zoomAndPan/index.htm`
[3]Available from `http://www.adobe.com/svg/`

Chapter 8

Command Reference

This section of the manual is intended to serve as a quick reference to ECharts commands and their command line options. More detailed explanations of the commands can be found elsewhere in the manual.

8.1 Machine Dependencies

ECharts translators determine dependencies amongst ECharts compilation units. For example, if a machine in one *.ech* file references a machine defined in another *.ech* file, then the default behavior for most translators is to automatically translate the other file as it translates the first file. This behavior can be disabled using the `--no-dependencies` command line option. Furthermore, if the modification date of an ECharts *.ech* file is later than the modification date of its respective ECharts host language file then the *.ech* file is re-translated to ensure the translated version is up to date. These aspects of ECharts translators are similar to the support provided by Java's `javac` compiler.

8.2 ECHARTSPATH

Another similarity between `javac` and ECharts translators is the ability to define a list of directories in which to search for compilation units. While `javac` uses the `CLASSPATH` environment variable, ECharts uses the `ECHARTSPATH` environment variable. When the ECharts translator needs to locate a compilation unit upon which another compilation unit depends, it searches each directory listed in the `ECHARTSPATH` variable in the order they are listed until it locates the compilation unit. Similar to `javac`, ECharts translators reference the `ECHARTSPATH` environment variable or the

`--echartspath` command-line argument. For example, in `Example0001` (see Section 2.1), the `ech2java` translator is invoked as follows:

```
% ech2java --echartspath .. Example0001.ech
```

However, an equivalent way to invoke the translator would be to initialize the `ECHARTSPATH` environment variable. Here we assume a Unix environment using the bash shell.

```
% export ECHARTSPATH=..; ech2java Example0001.ech
```

If the `ECHARTSPATH` environment variable is defined and an ECharts translator is invoked with the `--echartspath` command-line option, then the value defined by the command-line option will override the value defined by the environment variable.

8.3 `ech2java`

The `ech2java` translator translates ECharts *.ech* machine definition files to Java *.java* files. The `ech2java` translator is introduced in Section 2.1.

Usage:

```
ech2java [options] [echartsfilenames] [echartspackagenames]
```

Summary:

Translate specified `echartsfilenames` and `echartspackagenames` to Java *.java* files. For each specified package, translate all *.ech* files in that package.

Options:

--echartspath ':' separated list (';' on Windows) specifying directories to search for the specified `echartsfilenames` and `echartspackagenames`. The default value is the current working directory.

--no-dependencies Forces translator to not translate *.ech* files upon which the specified `echartsfilenames` and `echartspackagenames` depend. The default behavior is to translate dependencies.

--target-directory Directory path specifying where to write translated files to. If subdirectories on path do not exist then they are created. The default value is the source directory of the specified `echartsfilenames` and `echartspackagenames`.

--package-subdirectory Package subdirectory name relative to the target directory specifying where to write translated files to. The default value is the empty string.

--subpackages ':' separated list (';' on Windows) specifying package names recursively searched for *.ech* files. This is similar to the `echartspackagenames` command line argument except that package subdirectories are searched recursively for this option. The default value is the empty string.

--version Print translator version and exit.

--help Print help message and exit.

8.4 `ech2dot`

The `ech2dot` translator translates ECharts *.ech* machine definition files to Graphviz *.dot* files. The `ech2dot` translator is discussed in detail in Section 6.

Usage:

```
ech2dot [options] [echartsfilenames] [echartspackagenames]
```

Summary:

Translate specified `echartsfilenames` and `echartspackagenames` to Graphviz *.dot* files. For each specified package, translate all *.ech* files in that package.

Options:

--echartspath ':' separated list (';' on Windows) specifying directories to search for the specified `echartsfilenames` and `echartspackagenames`. The default value is the current working directory.

--no-dependencies Forces translator to not translate *.ech* files upon which the specified `echartsfilenames` and `echartspackagenames` depend. The default behavior is to translate dependencies.

--target-directory Directory path specifying where to write translated files to. If subdirectories on path do not exist then they are created. The default value is the source directory of the specified `echartsfilenames` and `echartspackagenames`.

--package-subdirectory Package subdirectory name relative to the target directory specifying where to write translated files to. The default value is the empty string.

--subpackages ':' separated list (';' on Windows) list specifying package names recursively searched for *.ech* files. This is similar to the `echartspackagenames` command line argument except that package subdirectories are searched recursively for this option. The default value is the empty string.

--label-formatter Fully qualified Python class name of label formatter. Default value is DotMachinePartialFormatter.DotMachinePartialFormatter.

--tooltip-formatter Fully qualified Python class name of tooltip formatter. Default value is DotMachineCommentFormatter.DotMachineCommentFormatter.

--url-formatter Fully qualified Python class name of URL formatter. Default value is DotMachineFormatter.DotMachineFormatter.

--version Print translator version and exit.

--help Print help message and exit.

8.5 `ech2doc`

The `ech2doc` translator translates ECharts *.ech* machine definition files to *.html* and *.svg* files. The `ech2doc` translator is discussed in detail in Section 7.2.

Usage:

```
ech2doc [options] [echartsfilenames] [echartspackagenames]
```

Summary:

Translate specified `echartsfilenames` and `echartspackagenames` to *.html* and *.svg* files. For each specified package, translate all *.ech* files in that package. Note that the Graphviz `dot` program must be installed to run this translator.

Options:

--echartspath ':' separated list (';' on Windows) specifying directories to search for the specified `echartsfilenames` and `echartspackagenames`. The default value is the current working directory.

--no-dependencies Forces translator to not translate *.ech* files upon which the specified `echartsfilenames` and `echartspackagenames` depend. The default behavior is to translate dependencies.

--target-directory Directory path specifying where to write translated files to. If subdirectories on path do not exist then they are created. The default value is the source directory of the specified `echartsfilenames` and `echartspackagenames`.

--subpackages ':' separated list (';' on Windows) list specifying package names recursively searched for *.ech* files. This is similar to the `echartspackagenames` command line argument except that package subdirectories are searched recursively for this option. The default value is the empty string.

--dot-path 'dot' program file path. Default value is 'dot'.

--document-title Documentation title HTML. Default value is 'ECharts Machine Documentation'.

--window-title Browser window title. Default value is document title.

--overview Path to documentation overview file shown on first page of documentation. Overview may utilize HTML tags. Default is no overview file.

--header Documentation footer HTML. Default value is empty string.

--svg-only Only generate *.html* and *.svg* files to support browsable SVG machine diagrams. Does not generate *.html* files for navigating machine packages or classes. Default behavior is to generate all *.html* files.

--svg-fontsize Font size in points for text associated with SVG machine diagrams. Default value is '8'.

--version Print translator version and exit.

--help Print help message and exit.

8.6 `ech2javadoc`

The `ech2javadoc` translator translates ECharts *.ech* machine definition files to *.html* and *.svg* files. The `ech2javadoc` translator is discussed in detail in Section 7.3.

Usage:

```
ech2javadoc [options] [echartsfilenames] [echartspackagenames]
```

Summary:

Translate specified `echartsfilenames` and `echartspackagenames` to *.html* and *.svg* files. For each specified package, translate all *.ech* files in that package. The target directory for translated files is a directory named *docfiles* that is a subdirectory of the *.ech* file source directory. Note that the Graphviz `dot` program must be installed to run this translator.

Options:

--echartspath ':' separated list (';' on Windows) specifying directories to search for the specified `echartsfilenames` and `echartspackagenames`. The default value is the current working directory.

--target-directory Directory path specifying where to write translated files to. If subdirectories on path do not exist then they are created. The default value is the source directory of the specified `echartsfilenames` and `echartspackagenames`.

--subpackages ':' separated list (';' on Windows) list specifying package names recursively searched for *.ech* files. This is similar to the `echartspackagenames` command line argument except that package subdirectories are searched recursively for this option. The default value is the empty string.

--dot-path 'dot' program file path. Default value is 'dot'.

--version Print translator version and exit.

--help Print help message and exit.

8.7 `javadocpp`

The `javadocpp` program post-processes HTML documentation generated by `javadoc` to provide customized views of ECharts machine class documentation. The `javadocpp` program is introduced in Section 7.3.

Usage:

```
javadocpp [options] [javadocdirs]
```

Summary:

Post-process specified `javadocdirs`. Each specified directory should contain `javadoc` HTML documentation. For each specified directory, post-process all HTML files corresponding to ECharts machine classes and recursively invoke the program for any subdirectories.

Options:

--version Print program version and exit.

--help Print help message and exit.

Chapter 9

Roadmap

Here's a non-prioritized overview of language and runtime features planned for future ECharts releases.

9.1 Exception Handling

The current version of ECharts requires programmers to handle runtime exceptions using the exception handling facility provided by the underlying host language. The next version will support exception handling in the ECharts language itself.

9.2 Machine Inheritance

Conspicuously absent from the current version of ECharts is support for machine inheritance. Our approach to inheritance will permit submachines to add or override states and transitions in machine superclasses.

9.3 Machine Variables

The adventurous ECharts programmer may have noticed that the current version of ECharts supports recursive machine invocation, that is to say, the ability for a machine to directly or indirectly invoke itself. With appropriately specified constructors and transition guards, one can even avoid infinite recursion. However this is more of a party trick than a fundamentally useful feature. What ECharts requires to make it a full-fledged (Turing complete) language is the ability to perform operations directly on a machine at runtime. Our approach will permit machines to be assigned to machine variables and machine variables to be assigned to machine states.

Our goal in providing this support is not to claim Turing completeness; rather, it is to simplify programming tasks we have encountered.

Appendix A

Building and Using ECharts

The ECharts SDK is supported on Solaris, Mac OS X, Linux, Windows and Eclipse. A number of additional open source software packages are required to build and use ECharts. We enumerate these packages here and provide instructions on how to configure your platform to use ECharts with these packages.

A.1 Software Requirements

The required software for building and using the ECharts SDK are Python 2.2 or greater and the Java 2 Platform, Standard Edition (JDK 1.4.2 or greater) and a suitable build tool. Python is bundled with the Linux and Mac OS X platforms. On Solaris and Windows a separate download is required from `http://python.org`. Java is bundled only with Mac OS X. Other platforms require a separate download from `http://java.sun.com`.

There are three possible build tools for building the ECharts SDK: the Eclipse SDK 3.1.2 or greater, Apache Ant 1.6.3 or greater, or GNU Make and the GNU BASH shell. Eclipse is available from `http://www.eclipse.org` for all platforms. Apache Ant is available from `http://ant.apache.org` for all platforms. GNU Make and BASH are bundled with Linux and Mac OS X (GNU Make is included with the Mac OS X Xcode Developer Tools). A separate download is required for Solaris from `http://gnu.org`. We do not recommend using Make or BASH on Windows since we have not tested them with the ECharts build. Instead, we recommend the use of Eclipse or Ant.

For generating machine diagrams or machine documentation (see Sections 6 and 7), Graphviz version 1.13 or greater is required. This is available as a separate download from `http://graphviz.org` for all platforms.

A.2 Building ECharts

The top-level build builds the ECharts language parser and the ECharts Java runtime library `echarts.jar`. This is a pre-requisite to translating ECharts machines, as described in Appendix A.3.

Before kicking off the build, ensure your platform's `PATH` environment variable references the Python executable directory. When building with Apache Ant, simply execute `ant all` in the ECharts top-level directory. When using Eclipse, select the 'all' target of the top-level *build.xml* file by invoking the file's 'Run As -> Ant Build...' contextual menu option. Note that Eclipse should be configured to use the external Java 2 JDK, not Ant's internal Java compiler. This is accomplished via the 'Java -> Installed JREs' node of the Eclipse 'Preferences' panel. When building with GNU Make, the value of the `JDK_HOME` environment variable must be set to the location of your JDK installation prior to invoking Make. One way to do this is from the BASH shell is as follows. Here we assume the JDK is installed in */usr/java/j2sdk1.4.2_10*.

```
% JDK_HOME=/usr/java/j2sdk1.4.2_10 make all
```

A.3 Using ECharts

Once the top-level build has been performed (see Appendix A.2) you are ready to start using the ECharts SDK to translate ECharts machines. The ECharts translators (e.g. `ech2java`) and associated programs (e.g. `javadocpp`) are Python scripts. They can be invoked either directly or indirectly.

A.3.1 Direct Invocation

On Unix platforms (Mac OS X, Solaris and Linux), the ECharts translators and programs can be invoked from the command line. To invoke a translator or program without having to prefix it with its path, the path to the translator executables can be added to the system path as follows. Here we assume the path to the ECharts SDK is *ECHARTS_HOME*, and that we are using the BASH shell.

```
export PATH=$PATH:ECHARTS_HOME/translator/bin
```

On Windows, you can invoke the ECharts translators and programs from the Windows `Cmd.exe` command line shell. However, invocations must be prefixed with an explicit invocation of Python. Here's an example

of invoking `ech2java` where we assume the path to the ECharts SDK is *ECHARTS_HOME*.

```
C:\> python ECHARTS_HOME\translator\bin\ech2java --version
```

A.3.2 Indirect Invocation

Translators and programs may also be invoked indirectly via Apache Ant on all platforms, or via GNU Make on Unix platforms (Mac OS X, Solaris and Linux). See the Ant and Make build files located in *runtime/java/src/examples* of the ECharts SDK for an example of invoking `ech2java`. Also see the Make and Ant build files located in *runtime/java/doc* in the ECharts SDK for an example of invoking `ech2javadoc` and `javadocpp`.

Appendix B

Licenses

All software in the ECharts SDK is open source software.

Unless otherwise indicated the software in the ECharts SDK is part of the ECharts package copyright (c) 2006 AT&T Corp. and is licensed under the Common Public License, Version 1.0 by AT&T Corp. (see `http://www.opensource.org/licenses/cpl1.0.php`).

The other license applicable to this SDK is the ANTLR 2 License (see `http://www.antlr.org/license.html`). This license covers the parser generator underlying ECharts machine translators.

B.1 Common Public License v1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS COMMON PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

   "Contribution" means:

   (a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and

   (b) in the case of each subsequent Contributor:

      i. changes to the Program, and

      ii. additions to the Program;

   where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution

'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents " mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

2. GRANT OF RIGHTS

   (a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.

   (b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

   (c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for

claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

(d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

(a) it complies with the terms and conditions of this Agreement; and

(b) its license agreement:

i. effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

ii. effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

iii. states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

iv. states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

(a) it must be made available under this Agreement; and

(b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

   EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

   If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed. In addition, if Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding com-

binations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. IBM is the initial Agreement Steward. IBM may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

B.2 ANTLR 2 License

We reserve no legal rights to the ANTLR–it is fully in the public domain. An individual or company may do whatever they wish with source code distributed with ANTLR or the code generated by ANTLR, including the incorporation of ANTLR, or its output, into commerical software.

We encourage users to develop software with ANTLR. However, we do ask that credit is given to us for developing ANTLR. By "credit", we mean that if you use ANTLR or incorporate any source code into one of your programs (commercial product, research project, or otherwise) that you acknowledge this fact somewhere in the documentation, research report, etc... If you like ANTLR and have developed a nice tool with the output, please mention that you developed it using ANTLR. In addition, we ask that the headers remain intact in our source code. As long as these guidelines are kept, we expect to continue enhancing this system and expect to make other tools available as they are completed.

Bibliography

[1] Gregory W. Bond. Timed transition activation semantics in Statecharts. Technical Report TD-6H4L5S, AT&T, 2005. Available from: `http://echarts.org`. 64

[2] Gregory W. Bond and Healfdene H. Goguen. ECharts: balancing design and implementation. In M.H. Hamza, editor, *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA 2002)*, pages 149–155. ACTA Press, 2002. Available from: `http://echarts.org`. 1

[3] Gregory W. Bond and Healfdene H. Goguen. ECharts: From lab to production. Technical Report TD-6FSMWT, AT&T, 2005. Available from: `http://echarts.org`. 1, 87

[4] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000. Available from: `http://graphviz.org`. 119

[5] Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot.* Available from `http://graphviz.org`. 122, 124

[6] Object Management Group. *OMG Unified Modeling Language Superstructure Specification, version 2.0.* Object Management Group, August 2005. Available from: `http://www.omg.org`. 1

[7] Thomas M. Smith and Gregory W. Bond. ECharts for SIP servlets: a state-machine programming environment for VoIP applications. In *IPTComm '07: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 89–98,

New York, NY, USA, 2007. ACM. Available from: http://echarts.org. 106