RC24387 (W0710-126) October 23, 2007
Computer Science

IBM Research Report

EventScript: Using Regular Expressions to Program Event-Processing Agents

Norman H. Cohen
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

EventScript: Using Regular Expressions to Program Event-Processing Agents

Norman H. Cohen

IBM Thomas J. Watson Research Center
Hawthorne, New York, USA
ncohen@us.ibm.com

Abstract. Large systems can be constructed out of nodes that consume and produce messages representing events. EventScript is a language for specifying the processing that is to take place at such a node. The underlying structure of an EventScript program is a regular expression for the sequence of events expected to be received. A major goal of EventScript was simplicity, so that the language would be easy to learn, efficient to execute, and easy to port to new target environments. Numerous event-processing examples illustrate EventScript programming techniques and the usefulness of EventScript for solving a wide variety of programming problems. We have developed a compiler, run-time environment, and testing tool for EventScript, as well as an API to facilitate the porting of EventScript to new execution environments. EventScript has been incorporated as one element of the DRIVE environment for developing, testing, deploying, and managing sensor/actuator applications.

1. Introduction

Unix introduced a simple but powerful model for coordination [Rit74]: Programs called *filters*, each reading from a standard input character stream and writing to a standard output character stream, can be chained together by *pipes* connecting the standard output of one filter to the standard input of another. Very simple filters, written independently of each other and communicating only through standard input and standard output, can be composed into powerful pipelines. Details of interleaving and buffering are handled automatically by the operating system. The ease of connecting filters with pipes encourages a style in which special-purpose filters with few behavioral variations can be composed in countless ways to offer a versatile set of functions.

Event processing networks, consisting of *event-processing agents* connected by *event channels* [Luc02], can be thought of as a generalization of filters and pipes. Unlike a filter, an event-processing agent may have many input channels and many output channels. Furthermore, the elements transmitted over the channel are not characters, but *event objects*, which may be elaborate data structures.

Event processing networks have been used for a wide variety of applications, such as physical process control, inventory control, and business monitoring. Potential sources of events include RFID readers, environmental sensors, point-of-sale terminals, and business-monitoring software. Potential consumers of events include signal lights, actuators ranging from process-control valves to door locks, graphical displays, messaging systems to send alerts to humans, and business-monitoring software.

Because of the narrow interfaces through which event-processing agents communicate, the design of an event-processing system can be decomposed into two distinct activities, reminiscent of DeRemer and Kron's programming-in-the large and programming-in-the-small [DeR75]. The first activity, corresponding to programming-in-the-large, is the design of the event processing network, in which event-processing agents with particular functional specifications are connected in a particular way by event channels. The second activity, corresponding to programming-in-the-small, is the specification of the behavior of individual event-processing agents. This paper describes a language, EventScript, that uses regular expressions to specify the behavior of individual event-processing agents.

Just as powerful and versatile pipelines can be composed from filters performing very narrow functions, powerful and versatile event-processing networks can be composed from event-processing agents performing very narrow functions. These functions include:

- event translation, for example converting physical units, reformatting dates, or dropping irrelevant fields
- composition of data from multiple event streams
- aggregation (e.g., summing or averaging) over successive events
- recognition a sequence of events matching a particular pattern, and emission of a compound event representing that instance of the pattern at a higher level of abstraction

It is a principal goal of EventScript to support the programming of such narrow functions. A small, easily learned, efficiently implementable set of primitives is sufficient to achieve this goal. When faced with a tradeoff between generality and simplicity, we have opted for simplicity. Even if this choice complicates the programming of the most intricate event-processing agents, we are confident that it facilitates the use of EventScript to program the most common event-processing agents. Because the programmers of event-processing agents are likely to be performing other tasks as well, involving other programming models and languages, they are best served by an unobtrusive notation that allows them to specify the behavior of the most common event-processing agents succinctly while learning few new concepts.

The programming community is familiar with the use of regular expressions to specify patterns of characters in text [Fri06]. This paradigm can be found in the awk, C#, Java, JavaScript Perl, PHP, Python, Ruby, and Visual Basic languages, for example, as well as in tools like grep, sed, and vi. Therefore, we expect that most programmers will be comfortable with the idea of using regular expressions to specify patterns of event occurrences in event streams.

To illustrate the basic structure of EventScript, we present an event-processing agent that averages readings from sensors S1, S2, and S3. Each time a new reading is received from any one of the sensors, the agent emits an event containing the newly computed average:

```
in double S1Input, double S2Input, double S3Input
out double Average
{ v1=0.0; v2=0.0; v3=0.0; }
(
   ( S1Input(v1) | S2Input(v2) | S3Input(v3) )
   { !>Average( (v1+v2+v3)/3.0 ); }
)*
```

The program begins by defining `S1Input`, `S2Input`, and `S3Input` to be names of input events (corresponding to readings arriving from S1, S2, and S3, respectively) and `Average` to be the name of an output event. Each of these events carries a value of data type `double`. Next comes the regular expression, within which are interleaved actions surrounded by curly braces. The regular expression consists of assignment actions to initialize variables `v1`, `v2`, and `v3`, which are eventually to hold the latest readings received from S1, S2, and S3, respectively, followed by a regular expression of the form `(...)*`, indicating repeated instances of the regular subexpression inside the parentheses. This regular subexpression consists of three alternatives (separated by the `|` operator) , followed by an emit action (denoted by the symbol `!>`). Each alternative matches a different kind of input event and saves the value carried by the event in the corresponding variable. The emit action emits an `Average` event carrying the value of the average of `v1`, `v2`, and `v3`. Besides matching event objects that arrive on an input event channel, EventScript can match events corresponding to the arrival of a specified time. In Section 2.4 we shall illustrate two variations on this example—one in which the event-processing agent reemits the current average if no average has been emitted in ten seconds, and one in which averages are emitted every ten seconds independently of the times that sensor readings arrive.

Besides making EventScript easy to learn, our preference for a sparse language over a general language makes EventScript amenable to a very small and fast run-time implementation, easily retargeted to new platforms. It will come as no surprise that, since an EventScript program is essentially a regular expression, it can be implemented essentially by doing a lookup in a state-machine transition table each time an event arrives. Efficiency and retargetability are important, because many event-processing applications involve real-time constraints and embedded processors with unique characteristics or limited resources.

The remainder of this paper is structured as follows: Section 2 is an overview of the features of the EventScript language. Section 3 addresses subtle issues in the language design. Section 4 shows EventScript at work, presenting a number of examples that illustrate a variety of EventScript programming techniques and idioms. Section 5 explains the connection between an EventScript program and real-world producers and consumers of event streams. Section 6 describes our implementation of EventScript, including the compiler, the run-time platform, and development tools. Section 7 describes various directions in which EventScript may evolve.

2. EventScript features

Before describing the features of EventScript, we explain the underlying notion of an event, and of the data carried by an event, in Section 2.1. The basic form of

an EventScript program is a sequence of declarations followed by a regular expression. Section 2.2 gives an overview of declarations. The most basic EventScript regular expression, an *event marker*, matches an input event with a specified name. EventScript regular expressions may be combined into larger regular expressions using the operators described in Section 2.3. Event markers themselves are discussed in Section 2.4. Section 2.5 describes the actions that may be interleaved in EventScript regular expressions. Section 2.6 describes how the value carried by an event can be used to determine dynamically the event name that will be assigned to that event, and thus which event markers that event will match. Section 0 describes how the values carried by events can be used to partition a stream of input events into substreams, each of which is matched independently and in parallel against the same regular expression.

2.1 Events and data types

An EventScript program receives a sequence of input events and emits a sequence of output events. Each input or output event has an event name and carries a value belonging to some EventScript data type. All input events of the same name carry values of the same data type, and all output events of the same name carry values of the same data type.

The EventScript language itself does not assume anything about the correspondence of input and output events to entities or occurrences in the outside world. Each individual EventScript run-time implementation establishes such a correspondence, as described in Section 5, but by keeping consideration of this correspondence out of the language itself, we make the language applicable in a wide variety of milieus. (Typically, the sequence of input events is a merged stream of input events arriving from different external sources, the name of an input event identifies the external source from which it arrived, each output event is delivered to an external destination identified by its event name, and there is a straightforward mapping between EventScript data types and the data formats of these external sources and destinations.)

EventScript has three kinds of data types—primitive data types, array data types, and structure data types. There are six primitive data types:

- `boolean` (true or false)
- `double` (an IEEE-754 double-precision floating-point value)
- `long` (a signed 64-bit integer)
- `string` (a string of Unicode characters)
- `time` (a timestamp corresponding to a particular date and time of day)
- `object` (an opaque object reference that can be taken from an input-event value or included in an output-event value, but whose internal structure cannot be accessed from within the EventScript language)

The values of an array data type are sequences of elements of a specified element data type, indexed starting at zero; the array data type with elements of data type t is denoted by t`[]`. The values of a structure data type are structures each having the same sequence of field names and data types; the structure data type with fields named n_1, ..., n_k, having data types t_1,...,t_k, respectively, is denoted as follows:

{ t_1 n_1; ... t_k n_k; }

(Events that serve as pure signals, carrying no information other than the fact that they occurred, carry values of the empty structure type, `{ }`.)

2.2 Declarations

There are four kinds of declarations—input-event declarations, output-event declarations, data-type definitions, and function declarations.

Input-event declarations and output-event declarations specify the names of input and output events, respectively, and the data types of the data they carry. For example, the input-event declaration

```
in double SetAlarmThreshhold,
   {string zoneName; double temperature;} ZoneReport
```

declares `SetAlarmThreshhold` to be an event name for input events carrying values of type `double`, and `ZoneReport` to be an event name for input events carrying values of a structure type with a field named `zoneName`, of type `string`, and a field named `temperature`, of type `double`; the declaration

```
out long PostSpeedLimit, {} CloseGate, {} OpenGate
```

declares `PostSpeedLimit` to be an event name for output events carrying values of type `long`, and `CloseGate` and `OpenGate` to be event names for output events that are pure signals. (Input-event declarations also have optional clauses, detailed in Sections 2.5 and 2.6, for specifying the features described in those sections.)

For convenience, and to facilitate abstraction, data types may be given names. The *data-type definition*

```
type Point = {double x; double y;}
```

allows the identifier `Point` to be used afterward as a synonym for the structure type `{double x; double y;}` any place where a data type is expected. In particular, the data-type name can be used in later data-type definitions. Thus an EventScript program beginning with the lines

```
type Point = {double x; double y;}
type Polygon = Point[]
in {string name; Polygon boundary;} SetZone
```

is equivalent to one beginning with just this line:

```
in {string name; {double x; double y;}[] boundary;} SetZone
```

Function declarations declare functions that are implemented outside of EventScript, so that they can be invoked by function calls in EventScript expressions. An external function might, for example, access system information such as an environment-variable value, or provide a domain-specific abstraction such as inclusion of a point in a polygon. The construct

```
function string envVar(string): "eslib.systemFuncs.EnvVar"
```

declares `envVar` to be an external function that takes a `string` argument and returns a `string` result. The string constant following the colon identifies the external implementation of the function.[1]

2.3 Regular-expression operators

Regular expressions may be combined into larger regular expressions using the conventional operators `*`, `|`, and sequence, as well as parentheses. Thus the regular expression *R*`*` matches any event sequence consisting of zero or more consecutive subsequences each of which matches regular expression *R*; the regular expression R_1 `|` ... `|` R_n matches any event sequence that matches at least one of the regular expressions R_1, ..., R_n; and the sequence R_1 ... R_n matches any event sequence consisting of *n* consecutive subsequences that match regular expressions R_1, ..., R_n in that order.

In addition, EventScript has several familiar extended regular-expression operators—the intersection operator `&`, the difference operator `-`, and several postfix *multipliers*, syntactically analogous to `*`: {*i*}, {*i*,*j*}, {*i*,}, `+`, and `?`, where *i* and *j* are numeric constants such that $0 \le i \le j$. The regular expression $R_1 \& R_2$ matches any event sequence that matches both regular expression R_1 and regular expression R_2. The regular expression R_1`-`R_2 matches any event sequence that matches regular expression R_1 but does not match regular expression R_2. If *R* is a regular expression, then *R*{*i*} matches an event sequence consisting of exactly *i* consecutive subsequences, each of which matches *R*; *R*{*i*,*j*} matches any event sequence consisting of from *i* to *j* such subsequences; *R*{*i*,} matches any event sequence consisting of *i* or more such subsequences (so that *R*{0,} is equivalent to *R*`*`), *R*`+` is equivalent to *R*{1,}, and *R*`?` is equivalent to *R*{0,1}).

2.4 Event markers

Just as the character `y` in the Perl string regular expression `x*yx*` is a placeholder matching an occurrence of the character `'y'` in a sequence of characters, the *event marker* `S1Input(v1)` in the EventScript regular expression of Section 1 is a placeholder for an occurrence of an event named `S1Input` in a sequence of input events. The event name in an event marker is followed by parentheses optionally containing a variable into which the value carried by the matched event should be stored.

In addition to the input events explicitly declared by input-event declarations, there are unnamed input events triggered by the passage of time. A time-triggered event carries a value of type `time`, specifying the time at which the event occurred.

---

[1] For the Java-based EventScript engine we have implemented, this string is the fully-qualified name of a Java class implementing the following interface:

```
public interface FunctionBody {
   EventScriptValue evaluate(EventScriptValue[] args)
      throws FunctionException;
}
```

An `EventScriptValue` object is the Java representation of an EventScript `boolean`, `double`, `long`, `object`, `string`, `time`, array, or structure value.

There are two special kinds of event markers for time-triggered events, one that is matched after a specified amount of time has elapsed and one that is triggered when a specified time (or a time matching a specified pattern) arrives.

The event marker `elapse[(x1+x2)/2 minutes](t)` is matched when `(x1+x2)/2` minutes have passed since the previous event (or since the beginning of program execution). The time at which the match occurs is captured in the variable `t`. The time unit in an `elapse` event marker may be days, hours, minutes, seconds, milliseconds, or microseconds[2]. Here is a variation on the average-of-three-sensors example of Section 1, in which the current average is emitted not only when data arrives from some sensor, but also when nothing has been emitted for the last ten seconds:

```
in double S1Input, double S2Input, double S3Input
out double Average

{ v1=0.0; v2=0.0; v3=0.0; }

(
  ( S1Input(v1)|S2Input(v2)|S3Input(v3)|elapse[10 sec]() )
  { !>Average( (v1+v2+v3)/3.0 ); }
)*
```

The event marker `arrive[2038-01-19 03:14:08]()` is matched at 14 minutes and eight seconds after 3:00 am on January 19, 2038.[3] The event marker `arrive[03:14:08](t)` is matched at that time every day, and the time of the match is captured in the variable `t`. The date/time pattern in an `arrive` event marker can also contain wildcards (denoted by a dot) in place of components of the date and time, so `arrive[.:00:00]()` is matched every hour on the hour and `arrive[.:.:00]()` is matched at the beginning of every minute. Any of the numbers in the date/time pattern can also be replaced by a parenthesized numeric expression, as in `arrive[(min/60):(min%60):00]()`. Finally, the entire date/time pattern can be replaced by a parenthesized expression of data type `time`, as in this variation on the average-of-three-sensors example, in which an average is emitted at regular ten-second intervals, independently of the times that sensor readings arrive:

```
in double S1Input, double S2Input, double S3Input
out double Average

{ v1=0.0; v2=0.0; v3=0.0; deadline=secondsAfter(10, now()); }

(    S1Input(v1)
   | S2Input(v2)
   | S3Input(v3)
   | arrive[(deadline)]()
        { !>Average( (v1+v2+v3)/3.0 );
          deadline = secondsAfter(10, deadline);
        }
)*
```

[2] Although values of type `time` can express timestamps to the nearest microsecond, the time-of-day clock that the EventScript run-time implementation uses for time-triggered events need not tick that often. The time-of-day clock in the Java implementation ticks every millisecond.

[3] The rules of EventScript do not address questions about the time-of-day clock such as the time zone in which it operates, the effect of a transition to or from daylight savings time, or the insertion of leap seconds into UTC. An EventScript implementation is free to address these issues in its own way.

(The built-in function `now()` returns the time value corresponding to the time it was invoked. The built-in function `secondsAfter(n,t)` takes a `long` value `n` and a `time` value `t` and returns the `time` value `n` seconds after the time denoted by `t`.)

Finally, there is a wildcard event marker, `.` (dot), that matches an event object of any *named* event type. Thus `.-S1Input()` matches an event object of every named event type other than `S1Input`, and `.*` matches an arbitrary sequence of named events. A wildcard event marker does not match a time-triggered event, so the following EventScript program emits a `Silence` event at the end of every one-minute period in which no event of one of the types `A`, `B`, ..., `Z` has arrived:

```
in {}A, {}B, ..., {}Z
out {}Silence

( . | elapse[1 minute]() { !>Silence({}); } )*
```

Since we do not know beforehand the name of the event that will matched by a wildcard, we do not know the data type of the value carried by the event. Therefore, unlike other event markers, a wildcard event marker does not include a parenthesized target variable for the value of the event.

2.5 Actions

An *action block*—a set of `{ ... }` braces surrounding a sequence of *actions*—may be placed before or after an element of an EventScript sequence regular expression. There are two kinds of primitive actions: emitting an output event object and assigning of the value of a Java-like expression to a variable. In addition, there are compound actions, including conditional actions, repeated actions, and nested action blocks. (Every primitive action ends with a semicolon.)

The *emit action* `!>Average( (v1+v2+v3)/3.0 );` emits an output event with the name `Average`, carrying the value of the expression `(v1+v2+v3)/3.0`. Given the output-event declaration

```
out {string employeeID; number exposure;} EmployeeExposure
```

the action

```
!>EmployeeExposure( {employeeID:"771803", exposure:10.3} );
```

emits an output event with the name `EmployeeExposure`, carrying a value that contains `"771803"` in its `employeeID` field and 10.3 in its `exposure` field. (The expression `{employeeID:"771803", exposure:10.3}` is an example of a *structure-builder*. Such an expression can also occur in contexts other than emit actions. The colons can be followed by arbitrary expressions for the field values.)

An *assignment action* computes a value and assigns it to a variable. The variable may be an identifier, a subscripted variable of the form *a*`[`*e*`]`, where *a* is a variable of some array type and *e* is an expression of type `long`, or a structure-field vari-

able of the form *s*`.`*f*, where *f* is an identifier and *s* is a variable of some structure type having a field named by *f*.[4]

In the spirit of scripting languages, EventScript variables are not declared. However, they must be used in a type-consistent manner. For example, the same variable may not appear both in a context requiring a `long` variable and a context requiring a `boolean` variable. The EventScript compiler infers the types of variables, starting with the data types of variables in event markers (as specified in the input-event declarations with the same event name), the expressions in emit actions (as specified in the output-event declarations with the same event name), and the type requirements of certain syntactic contexts (such as the requirement that a condition in a conditional action be of type `boolean`, or that the subscript in a subscripted variable be of type `long`). The EventScript compiler flags type inconsistencies and also warns about variables potentially set but never read, or read without having been set. In the absence of explicit variable declarations, such warnings are essential to catch misspellings of variable names.

Additional aspects of EventScript actions are illustrated by the following program, which emits smoothed averages of sensor readings. The program initially accepts a `SetWindowSize` event that specifies the size of the window used for computing sliding averages, say *N*. Then it accepts `Reading` events containing actual sensor data. Starting when *N* sensor readings have been received, each `Reading` event is followed by the emission of an `Average` event containing the mean of the *N* most recent sensor readings:

```
in long SetWindowSize, double Raw
out double Average

SetWindowSize(N)
{ history = new double[N];
  for (i: 0, N-1) history[i] = 0.0;
  sum = 0.0;
  readingCount = 0;
  circularBufferCursor = 0;
}
(
   Raw(newValue)
   {  sum = sum - history[circularBufferCursor] + newValue;
      readingCount = readingCount + 1;
      readingCount>=N ? !>Average(sum/N);
      history[circularBufferCursor] = newValue;
      circularBufferCursor = (circularBufferCursor+1) % N;
   }
)*
```

As in Java, the expression `new double[N]` allocates an array with `N` uninitialized elements of type `double`. The *repeated action*

```
for (i: 0, N-1) history[i] = 0;
```

[4] If the variable `v` belongs an array or structure data type, the assignment "`w=v;`" stores a reference to a shared array or structure into `w`, so that the subsequent assignment "`v[0]=3;`" or "`v.f=3;`", respectively, causes `w[0]` or `w.f`, respectively, to assume the value `3`.

executes the action `history[i] = 0;` with `i` set to successive values in the range zero to `N`. The *conditional action*

```
readingCount>=N ? !>Average(sum/N);
```

executes the action `!>Average(sum/N);` if and only if the condition

```
readingCount>=N
```

evaluates to *true*.[5] The array `history` acts as a circular buffer for the *N* most recent readings. Rather than sum the contents of `history` each time an average is computed, we maintain that sum in the variable `sum`, and use the buffer only to determine the amount to subtract from `sum` when the oldest reading in the buffer is replaced.

A function call within an expression may call an EventScript built-in function or an external function implemented outside of EventScript. An external function implemented outside of EventScript can only be called if it is declared by a function declaration, as described in Section 2.2.

2.6 Event classification

We would sometimes like the role that an occurrence of a particular kind of event plays in matching a pattern to depend on the value carried by the event occurrence. EventScript provides for *classifying* events arriving with a particular name—that is, assigning them new event names—depending on the value carried by each event. An event marker may refer to the new name. For example, the input-event declaration

```
in double Temperature
   case { Temperature >= 100.0 ? High : Low }
```

contains an *event-case clause* specifying that input events named `Temperature` carry values of type `double`, but that those arriving with values of at least 100.0 should be renamed with the event name `High` and the remainder should be renamed with the event name `Low`. Within a condition of the event-case clause (in this example, "`Temperature >= 100.0`"), the original name of the arriving event (`Temperature`) represents the value carried by the event to be classified.

Event classification can be used for filtering a stream of input events. The following event script program accepts a stream of Temperature input events, discards those with values below 100, and emits a `FilteredTemperature` event corresponding to each Temperature event with a value greater than or equal to 100:

[5] As in many programming languages, if one operand of a binary operator is of type `long` and the other is of type `double`, the long operand is "promoted" to type `double`. In this example, `N` is promoted to type `double` and a floating-point division is performed. However, since the data type of the target variable in an EventScript assignment action is inferred from the context rather than specified in a declaration, EventScript does not perform implicit conversions of assigned values. An assignment such as "`N=sum;`" or "`sum=N;`" in this example would result in a type error. The programmer must explicitly write "`N=long(sum);`" or "`sum=double(N);`" (calling one of the built-in type-conversion functions `long` or `double`) to assign a converted value.

```
in double Temperature
   case { Temperature >= 100.0 ? High : Low }

out double FilteredTemperature

( High(x) { !>FilteredTemperature(x); } | Low() )*
```

The following EventScript program emits an `Alert` event if a temperature reading of 100 degrees or more is received and no temperature reading lower than 100 degrees is received for an hour afterward; the program emits a `Clear` event the first time a temperature reading lower than 100 degrees is received after an `Alert` event has been issued.

```
in double Temperature
   case { Temperature >= 100.0 ? High : Low }

out {} Alert, {} Clear

( // in cleared state
  Low()*
  High() { alertTime = hoursAfter(1,now()); }
  // in pending state
  High()*
  (   Low()
    |
      arrive[(alertTime)]() {!>Alert( {} );}
      // in alerted state
      High()*
      Low() {!>Clear( {} );}
  )
  // back in cleared state
)*
```

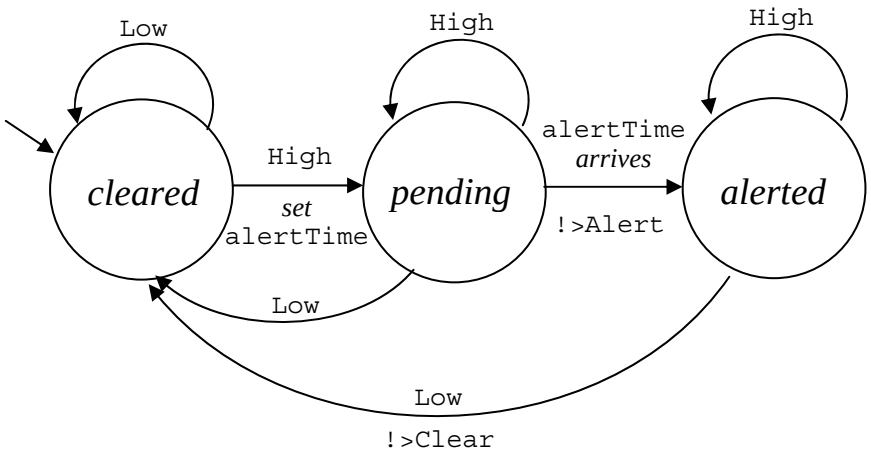

Fig. 1. A state-machine representation of the EventScript program that issues alerts about sustained high temperatures.

(As illustrated in Fig. 1, this program begins in a *cleared* state, in which it accepts `Low` events without performing any action. When the program receives a `High` event while in the cleared state, it sets `alertTime` to the time at which an `Alert` event

should be issued if no `Low` event arrives in the meantime, and enters a *pending* state. While in the pending state, the program accepts further `High` events without performing any action. The program stays in the pending state until either it receives a `Low` event—in which case the program returns to the cleared state—or `alertTime` arrives—in which case the program emits an `Alert` event and enters an *alerted* state. In the alerted state, the program accepts further `High` events without performing any action, but when it receives a `Low` event, the program emits a `Clear` event and returns to the cleared state.)

2.7 Event grouping

Event grouping is the splitting of a stream of events into substreams according to the values carried by the events, and the matching of patterns independently within each substream. Event grouping is sometimes called *event correlation*. For example, a program monitoring suspicious stock-trading patterns might issue an alert if the same trader executes three or more trades valued over $10,000 for the same company in a space of one hour. The alert is issued for the third and any subsequent trades within any one hour span, and includes the average value of the three most recent trades over $10,000. A pattern matching three trades valued over $10,000 in a space of one hour should be processed independently for each combination of trader and company. Fig. 2 depicts the processing that should be performed for a given trader and company.

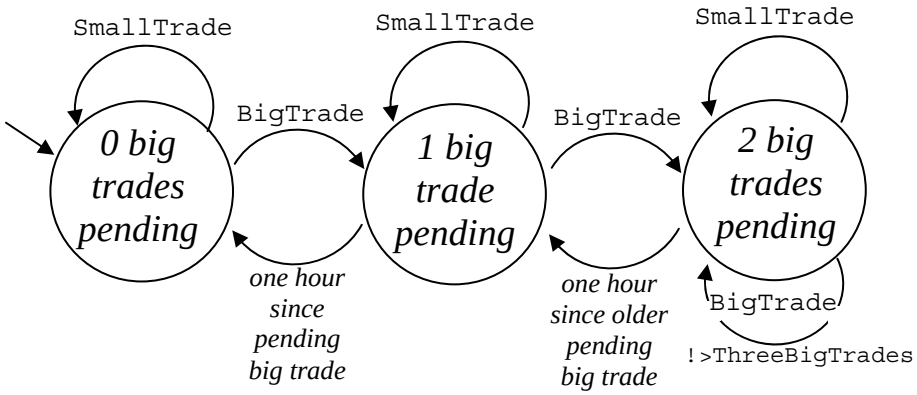

Fig. 2. A state-machine representation of the EventScript pattern for a given stock and trader that issues an alert when there have been three or more big trades of that issue by that trader within one hour.

The following EventScript program solves this problem:

```
in { string traderID;
     string company;
     double price;
     long volume;
     time when;
   } Trade
   group(Trade.traderID, Trade.company)
   case { Trade.price*Trade.volume>=10000.0 ?
              BigTrade
            : SmallTrade
        }
```

```
out {string traderID; string company; double averageValue;}
       ThreeBigTrades

 (
   // 0 big trades pending

   SmallTrade()*
   BigTrade(bt1)
   { expiration1 = hoursAfter(1,bt1.when);
     value1 = bt1.price * bt1.volume;
   }

   // 1 big trade pending

   (
      SmallTrade()*
      BigTrade(bt2)
      { expiration2=hoursAfter(1,bt2.when);
        value2 = bt2.price * bt2.volume;
      }

      // 2 big trades pending

      (
         SmallTrade()*
         BigTrade(bt3)
         { value3 = bt3.price * bt3.volume;
           outputValue =
              { traderID: bt3.traderID,
                company: bt3.company,
                averageValue: (value1+value2+value3)/3 };
           !>ThreeBigTrades(outputValue);
           value1 = value2;
           value2 = value3;
           expiration1 = expiration2;
           expiration2 = hoursAfter(1,bt3.when);
         }
      )*
      arrive[(expiration1)]()
      { expiration1 = expiration2;
        value1 = value2;
      }

      // 1 big trade pending

   )*
   arrive[(expiration1)]()

   // 0 big trades pending
)*
```

The line

```
group(Trade.traderID, Trade.company)
```

in the input declaration for `Trade` is a *group clause*, indicating that incoming events will be grouped according a key with two components. In this case, the first component is the `traderID` field of the value carried by the arriving `Trade` event and the second component is the `company` field of that value; in general, the group-clause expressions may involve arbitrary computations based on the value carried by the arriving event. For every resulting key value, there will, in effect, be a distinct instance of the state machine depicted in Fig. 2. In a program with group clauses, every input-event declaration must have a group clause, each group clause must have the same number of key components, and corresponding key components must have the same data type in each group clause. Thus each incoming `Trade` event is directed to an instance of the pattern matching process determined by the values of its `traderID` and `company` attributes.

(The assignment actions in this program maintain the following invariants:

- When one or two big trades are pending, the variable `expiration1` holds the expiration time for the oldest pending big trade and the variable `value1` holds the value of that trade.
- When two big trades are pending, the variable `expiration2` holds the expiration time for the more recent pending big trade and the variable `value2` holds the value of that trade.

When two big trades are pending and another big trade arrives, the trade that had been the more recent becomes the oldest and the one that just arrived becomes the more recent pending trade. When two big trades are pending and the older one expires, the trade that had been the more recent pending trade becomes the oldest pending trade.)

The `traderID` and `company` components of the structure value emitted in a `ThreeBigTrades` event reflect the corresponding components of the grouping key for the current instance of the state machine. In this program, the values of the grouping-key components were obtained from the value `bt3` carried by the most recent `BigTrade` event. However, there are other programs that require knowledge of a grouping-key component value in contexts in which this information is not available from a previously received input event. An expression of the form `group[`*n*`]`, where *n* is a zero-based index into the components of the grouping key, gives the value of the corresponding grouping-key component for the current instance of the state machine.

3. Semantic issues

Section 2 explained the fundamental features of EventScript, and the information provided there is sufficient to begin writing useful EventScript programs. In this section we discuss that do not play a central role in day-to-day programming, but which presented interesting challenges in the design of the language. Section 3.1 explains how EventScript deals with runtime errors. Section 3.2 explains how the notion matching a regular expression, which has traditionally been applied to strings of a known length, applies to a stream of incoming events that may have no end. Section

3.3 explains how EventScript deals with programs that may match an event stream in more than one way.

3.1 Runtime errors

A number of errors can arise during the execution of an EventScript program:

- the arrival of an event whose name was not declared as an input-event name
- the arrival of an event whose name was declared, but which does not match the regular expression
- an attempt to evaluate an uninitialized variable
- a value of zero for the right operand of the division or remainder operator for type `long`[6]
- a subscript out of bounds

In any of these cases, the EventScript program emits an implicitly declared output event whose name is the reserved name "`$error`" and whose value is a structure containing a numeric error code and a textual description of the error. For errors involving the arrival of an unexpected event, the offending event is discarded. For errors involving the evaluation of an expression, a default value (depending on the type of the expression) is used. Execution then proceeds as normal. As Section 5 will explain, the handling of emitted error events depends on the EventScript run-time implementation.

3.2 Termination, acceptance, and rejection

The following EventScript program receives a `Start` event, followed by zero or more `Ping` events, followed by a `Stop` event, then emits a `PingCount` event reporting the number of `Ping` events it received, then terminates:

```
in {} Start, {} Ping, {} Stop
out long PingCount

Start() { n = 0; }
( Ping() { n = n+1; } )*
Stop() { !>PingCount(n); }
```

In contrast, the following EventScript program does not terminate, but repeatedly waits for packets of events consisting of a `Start` event, followed by zero or more `Ping` events, followed by a `Stop` event, and emits a `PingCount` event at the completion of each packet reporting the number of `Ping` events in that packet:

```
in {} Start, {} Ping, {} Stop
out long PingCount

( Start() { n = 0; }
  ( Ping() { n = n+1; } )*
  Stop() { !>PingCount(n); }
)*
```

[6] For type `double`, division by zero produces an IEEE 754 negative or positive infinity result and remainder mod zero produces an IEEE 754 NaN (Not a Number) result.

There are also programs that may or may not terminate, depending on the input data, such as this one:

```
in boolean TerminationSwitch
      case { TerminationSwitch ? DoOnce : DoRepeatedly }

in {} Start, {} Ping, {} Stop

out long PingCount

  DoOnce()
  // Match one packet, then terminate:
  Start()
  { n = 0; } ( Ping() { n = n+1; } )*
  Stop() { !>PingCount(n); }
|
  DoRepeatedly()
  // Match zero or more packets, and do not terminate:
  ( Start() { n = 0; }
    ( Ping() { n = n+1; } )*
    Stop() { !>PingCount(n); }
  )*
```

As Section 0 explained, the execution of a program with event grouping consists, in effect, of a separate instance of the computation for each value of the grouping key. When a particular grouping-key value is first encountered, the EventScript run-time environment allocates resources for the computation instance corresponding to that key. In the case of a program with a terminating regular expression, the EventScript run-time environment releases those resources when the computation instance terminates. However, in the case of a program with a nonterminating regular expression, the resources are never released. If the set of possible grouping keys is relatively small—for example, if each grouping key corresponds to a different room in a particular building—the accumulation of computation instances presents no problem. However, if the set of possible grouping keys is open-ended—for example, if each grouping key corresponds to an account number—then the accumulation of computation instances constitutes an unsustainable resource leak. In an application with an open-ended set of possible grouping keys, care should be taken to write terminating regular expressions. (Our EventScript compiler warns the programmer about potentially nonterminating programs with grouping keys.)

The traditional definition of a finite automaton, acting on a string of symbols of some known length, designates each state as either an accepting state or a nonaccepting state; the finite automaton either accepts the string (if the transitions induced by the symbols of the string leave it in an accepting state) or rejects it (if they leave the automaton in a nonaccepting state). In the case of an EventScript program acting on a stream of events, we do not always know whether we have seen all events in the stream, so we are not interested in whether the corresponding finite automaton is in an accepting state following the sequence of events we have seen so far. Rather, we are interested in whether the sequence of events we have seen so far is one that could be extended in a way that matches the program's regular expression. For example, given the regular expression `( A() B() )*`, we do not distinguish between the validity of the computation after receiving the event sequence `A`, `B` (in which the regular expres-

sion has been matched) and after receiving the event sequence `A` (in which the regular expression may be on its way to being matched). However, as soon as we receive an event that cannot possibly be part of a matching string (say an `A` event following another `A` event), we signal a run-time error as described in Section 3.1.

3.3 Ensuring deterministic execution

3.3.1 Ambiguous regular expressions

An EventScript program may be *ambiguous*. That is, for a given sequence of input events, there may be multiple paths through its regular expression, possibly with different actions along each path.[7] For example, an `A` event may match either alternative in the program of Fig. 3, and the value carried by the emitted `B` event depends on which path is chosen. Ultimately, an ambiguity in a regular expression is attributable to one of two circumstances:

- The next arriving event can be interpreted as matching more than one alternative in a set of alternatives, as in Fig. 3.
- In a sequence regular expression containing a repetition, the next arriving event can be interpreted as matching either the beginning of the repetition or the beginning of the part of the sequence following the repetition, as in the regular expression `( A() B() )* A() C()`.

Sometimes an ambiguous regular expression is considerably more succinct than an equivalent unambiguous regular expression, and sometimes there is not an equivalent unambiguous regular expression. Thus, EventScript allows ambiguous regular expressions, but our compiler issues warnings when it encounters them.

```
in {} A
out long B

( A() {x=1;} | A() {x=2;} ) {!>B(x);}
```

Fig. 3. An ambiguous EventScript program that executes assignment actions on different paths in response to the same input event.

The execution of an EventScript program *follows all feasible paths simultaneously*; on a transition that causes actions to be reached on multiple paths, all actions reached are executed, in order of their textual occurrence in the program. Thus, when the program in the previous paragraph receives an `A` event, it executes first the action

[7] Book et al. [Boo71] call a regular expression *unambiguous* if, for all input sequences matching the regular expression there is exactly one mapping from symbols in the input sequence to matching occurrences of symbols in the regular expression. For every regular expression that is ambiguous in this sense, there is an unambiguous regular expression matching the same set of sequences. Bruggemann-Klein and Wood [Bru94] call a regular expression *1-unambiguous* if it is always possible to determine the symbol occurrence in the regular expression that should match the next input symbol by looking only at the input symbols that have already been read and the next input symbol. That is, a regular expression is 1-unambiguous if it can be matched in a unique manner with one-symbol lookahead. Some regular expressions have no 1-unambiguous equivalents. When we refer to an *ambiguous* EventScript program, we mean one whose regular expression is not 1-unambiguous in the sense of Bruggemann-Klein and Wood.

`x=1` and then the action `x=2`, leaving `x` holding the value 2 when the action `!>B(x);` is executed. In contrast, the program of Fig. 4 responds to an `A` event by first executing the action `!>B(1);` and then executing the action `!>B(2);`. The all-feasible-paths semantics follows naturally from taking a nondeterministic finite automaton reflecting an ambiguous regular expression and applying the classic construction of a deterministic finite automaton from a nondeterministic one [Rab59], in which there is a one-to-one correspondence between the state of the deterministic finite automaton reached for a given input and the *set* of states of the nondeterministic finite automaton reached for that input.

```
in {} A
out long B

( A() {!>B(1);} | A() {!>B(2);} )
```

Fig. 4. An ambiguous EventScript program that executes emit actions on different paths in response to the same input event.

The intersection regular-expression operator `&` and the difference regular-expression operator `–` work by following all feasible paths through both operands. A sequence of events matches an instance of the intersection operator if and only if it has at least one feasible path through each operand; it matches an instance of the difference operator if and only if it has at least one feasible path through the left operand and no feasible path through the right operand. Interestingly, in a regular expression of the form *L* `–` *R* `{` *a* `}`, where *L* and *R* are regular subexpressions and a is an action (or sequence of actions) executed when *R* has been matched, *a* is executed only for event sequences that do not match the difference regular expression as a whole. In effect, *a* is a response to the discovery of a reason that the event sequence should not be matched by the regular expression as a whole.

EventScript is intended, in part, for responsive applications in which it is not feasible to defer actions until later input events resolve ambiguities. Therefore, we perform an action as soon as it is reached: The program

```
in {} A, {} B, {} C
out {} Z

( A() B() | A() {!>Z({});} C() )
```

emits a `Z` event as soon as an `A` event arrives, rather than waiting for the next event to determine which alternative should have been selected. Furthermore, since an action can have an irreversible external side effect—the emitting of an output event—we cannot backtrack when the arrival of a later event EventScript establishes that one of the paths taken was wrong. Even if the arrival of the `A` event is followed by the arrival of a `B` event, a `Z` event has already been emitted—and possibly processed by a system that has performed some physical action in response—and cannot be retracted.

This “greedy” execution of actions arises not only in alternatives, but also in repetitions. After the program

```
in {} A, {} B, {} C
out {} Y, {} Z

A() ( { !>Y({}); } B() )* { !>Z({}); } C()
```

encounters an A event, it is poised to iterate the repetition zero or more times. Since the emit action for Y and the emit action for Z are both reachable, both actions are executed greedily at this point (and, similarly, after each subsequent arrival of a B event). If the intent is to execute a Y action only on each actual execution of the body of the repetition, and to execute a Z action only after all repetitions are done, the regular expression should be written as follows instead:

```
A() ( B(){ !>Y({}); } )* C(){ !>Z({}); }
```

The general principle is that actions should be "guarded" by event markers for the events whose arrivals are meant to trigger the actions.[8]

Much of the expressive power of regular expressions comes from ambiguity. For example, suppose we have a keypad with keys labeled 0 through 9, we receive a Key event carrying the value *x* whenever the key labeled *x* has been pressed, and we wish to emit an Unlock event whenever the key sequence 0, 0, 7 is pressed in the middle of an arbitrary sequence of key presses. The following EventScript program solves this problem in a straightforward way:

```
in long Key case { Key==0 ? K0 : Key==7 ? K7 : Other }

out {} Unlock

( .* K0() K0() K7() { !>Unlock({}); } )*
```

The regular expression in this program is ambiguous. For example, there are three ways the sequence of key presses 1, 0, 0 can begin to match it:

- The repeated wildcard `.*` matches all three key presses.
- The repeated wildcard `.*` matches the first two key presses and the first event marker `K0()` matches the third key press.
- The repeated wildcard .* matches the first key press and the sequence of event markers `K0() K0()` matches the second and third key presses.

The EventScript all-feasible-paths semantics has exactly the desired effect, causing an Unlock event to be emitted precisely when a sequence of key presses ending in 0, 0, 7 has been entered. The following unambiguous EventScript program has the same behavior, but it is both longer and less obvious:

```
in long Key case { Key==0 ? K0 : Key==7 ? K7 : Other }

out {} Unlock
```

[8] The regular expression `( A()* { !>Z(); } )*` has a path with an infinite number of action executions, with no intervening input event. (This path corresponds to zero iterations of the inner repetition on each iteration of the outer repetition.) Greedy execution of actions would imply that the EventScript program should immediately enter an infinite loop emitting Z events. However, EventScript prohibits (and our EventScript compiler detects) any program containing a cyclic path that has actions but no event marker.

```
(
   // No zeroes pending
   ( K7() | Other() )*
   // No zeroes pending
   K0()
   // One zero pending
   ( ( K7() | Other() )+ K0() )*
   // One zero pending
   K0()+
   // Two zeroes pending
   ( K7() { !>Unlock({}); } | Other() )
   // No zeroes pending
)*
```

For both programs, our EventScript compiler generates a deterministic finite automaton equivalent to the one shown in Fig. 5.

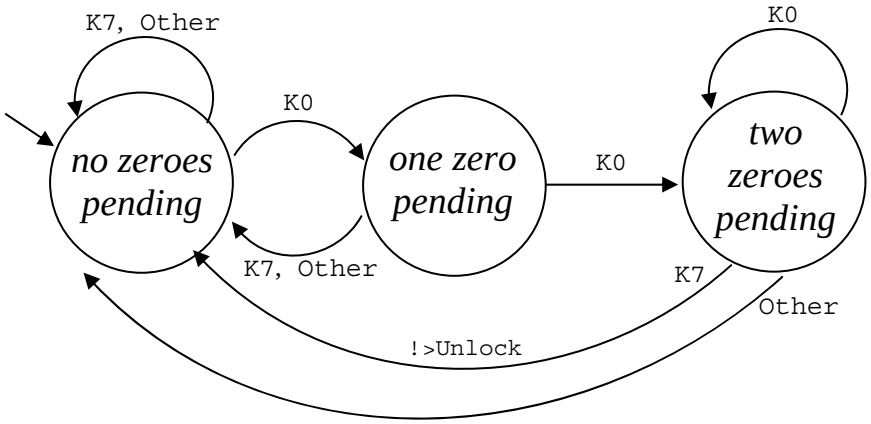

Fig. 5. A deterministic finite automaton that emits an **`Unlock`** event whenever it has received a sequence of input events ending with **`K0`**, **`K0`**, **`K7`**.

There are also ambiguous regular expressions for which there is no unambiguous equivalent. For example, the program

```
in {} A, {} Even, {} Odd

out {} EvenSeen, {} OddSeen

( A() A() )*
( Even() {!>EvenSeen({});} | A() Odd() {!>OddSeen({});} )
```

—which matches either an even number of `A` events followed by an `Even` event or an odd number of `A` events followed by an `Odd` event—is ambiguous because, after an even number of `A` events (possibly zero) have been seen, the next `A` event to arrive may match either the first `A()` event marker in the repetition or the `A()` event marker at the start of the second alternative following the repetition. It can be shown by the methods of [Bru98] that *any* regular expression that is built using `|`, `*`, and sequence operations, and that matches either an even number of `A` events followed by an `Even` event or an odd number of `A` events followed by an `Odd` event, is necessarily ambiguous. In this case, there is no harm done by following both feasible paths simultaneously, because the arrival of the next event after the problematic `A` event renders one of the paths infeasible: If that event is another `A` event, we must have been on the path

through the repetition; if it is an `Odd` event, we must have been on the path through the second alternative. Importantly, the emit actions are executed at points where there is only one feasible path through the regular expression (because we have seen either an `Even` event or an `Odd` event).

In general, an ambiguous regular expression is innocuous as long as the set of feasible paths through the regular expression has collapsed back into a single path at all points where an action is reached. (Like the even/odd program, the ambiguous version of the keypad program has this property. The programs of Fig. 3 and Fig. 4 do not.) However, the execution of actions on different paths in response to a single input event can also be innocuous. For example, suppose a retailer wishes to reward a customer with a discount coupon after every third purchase, and with free software after every computer-related purchase. The following program issues output events signaling the granting of a coupon or software in response to input events reporting purchases:[9]

```
in {string customerID; boolean isComputerRelated;} Purchase
   group(Purchase.customerID)
   case { Purchase.isComputerRelated ?
             CompPurchase
          : OtherPurchase
        }

out string GiveCoupon, string GiveSoftware

   ( Purchase(p){3} { !>GiveCoupon(p.customerID); } ) *
|
   (   CompPurchase(cp) { !>GiveSoftware(cp.customerID); }
     |
       OtherPurchase()
   )*
```

In effect, the two alternatives at the outermost level of this regular expression are computations preceding in parallel: For each input event, the program follows one path through the top alternative and one path through the bottom alternative. When a customer's third purchase is computer-related, the customer should be rewarded with both a coupon and software; the program reflects this by emitting a `GiveCoupon` event followed by a `GiveSoftware` event (in that order, based on lexical position) as both emit actions are encountered in response to the corresponding `Purchase` event. This paradigm is reminiscent of *orthogonal products* in Statecharts [Har87], in which the state of a box consists of the states of several contained boxes responding independently to each input.

In the customer-reward example, the actions reached along different paths through the regular expression in response to the same input manipulate disjoint sets of variables. The behavior of a program becomes much harder to understand if the actions along one path read or modify a variable modified along the other path. The rule stipulating that such actions are executed in order of lexical occurrence in the

[9] The presence of an event-case clause in the declaration of an event name enables the writing of event markers with the event names in the event-case clause, but it does not preclude the writing of event markers with the original, unclassified event name. In this example, the event marker `Purchase(p)` is equivalent to `( CompPurchase(p) | OtherPurchase(p) )`.

program makes the behavior of such a program deterministic: Race conditions of the sort arising in multithread programs do not arise. Nonetheless, we do not encourage programming styles that depend subtly on lexical order.

This section has highlighted examples in which ambiguity is useful, to justify our decision to allow ambiguous regular expressions in EventScript. However, in our experience, the majority of real-world event-processing problems are most naturally solved with unambiguous regular expressions. Typically, an EventScript programmer need not be concerned with the rules for ambiguous regular expressions. In the unusual case where an ambiguity does arise, our EventScript compiler calls the programmer's attention to this fact with a warning message.

3.3.2 Ambiguity arising from simultaneously firing event markers

Section 3.3.1 was concerned with ambiguity about the current position in a regular expression after a specified sequence of events has occurred. A different kind of potential ambiguity concerns which events have occurred. This form of ambiguity arises when multiple `elapse` or `arrive` event markers fire simultaneously. Whether or not time-event markers actually fire simultaneously depends, in general, on the values of expressions appearing in the markers. While our EventScript compiler warns about the *potential* for simultaneously firing time-event markers, the actual discovery of such an event would require a run-time check. Rather than performing a run-time check, we define the ambiguity away by arbitrarily choosing, from among simultaneously firing time-event markers, that which occurs earliest in the text of the program.[10] For example, the program

```
out {} StartOfMinute, {} StartOfHour

(   arrive[.:00:00]() {!>StartOfHour({});}
  | arrive[.:.:00]() {!>StartOfMinute({});}
)*
```

emits a `StartOfMinute` event at the start of every minute, except for the first minute of every hour, when the earlier `arrive` event marker is selected and a `StartOfHour` event is emitted *instead*. (Were we to reverse the order of the alternatives, the `StartOfMinute` event would be emitted every minute, and the `arrive` event marker associated with the `StartOfHour` event would never be selected.) Of course there is a less tricky way to achieve this effect, not depending on the order-of-occurrence tie-breaking rule:

```
( arrive[.:00](t)
  { minute(t)==0 ? !>StartOfHour({}); : !>StartOfMinute({}); }
)*
```

[10] In our implementation, this choice is made upon transition to a state in which multiple time-event markers become eligible to be matched next: We evaluate the expressions in each of these time-event markers, determine which marker has the earliest deadline, and use textual position to break ties. A wakeup timer is set only for the textually earliest of the time-event markers sharing the earliest firing time.

4. EventScript programming techniques

The use of regular expressions annotated with actions to process event streams is a very flexible paradigm, enabling a wide variety of programming idioms and styles. This section discusses some of those we have discovered so far. Of course additional EventScript programming techniques will be discovered over time.

4.1 Modeling the input stream

An EventScript regular expression can be thought of as a model of the stream of events that will arrive during the lifetime of the program. This model reflects not only the event sequences for which actions should be performed, but also the event sequences that should be ignored.

Consider, for example, the problem of ensuring that a jet engine is inspected after each use. A `Use` event arrives when the engine is started and an `Inspection` event arrives when completion of an inspection is detected. A `Violation` event is to be emitted whenever two `Use` events are detected without an intervening `Inspection` event. It is tempting to attempt to solve this problem with the following program:

```
// Version 1:

in {} Use, {} Inspection
out {} Violation

( Use() Use() { !>Violation({}); } )*
```

However, as Section 3.1 explained, the arrival of an input event that does not match the regular expression of the program results in a run-time error. This program will generate an error every time an `Inspection` event arrives. Worse, the unexpected `Inspection` event will be discarded and normal execution will continue after an error event is emitted, so an `Use` event followed by a (discarded) `Inspection` event followed by an `Use` event will incorrectly cause a `Violation` event to be emitted.

Since many regular expressions can be written to match the same set of input sequences, there are many ways to model the expected set of input events. The following version of the program works correctly:

```
// Version 2:

in {} Use, {} Inspection
out {} Violation

{ inspectionNeeded = false; }
```

```
(   Use()
    { inspectionNeeded ?
           !>Violation({});
         : inspectionNeeded = true;
    }
  |
    Inspection()
    { inspectionNeeded = false; }
)*
```

So, however, does the following version:

```
// Version 3:

in {} Use, {} Inspection
out {} Violation

(   Use()
    ( Use() { !>Violation({}); } )*
    Inspection()
  |
    Inspection()
)*
```

Version 2 is a generic model that says very little about the underlying structure of the input stream. If we ignore the actions, we are left with a regular expression that says simply that we expect a stream of zero or more input events, each of which is either a `Use` event or an `Inspection` event. The actions following the `Use` and `Inspection` event markers specify how each kind of input event is to be handled. In fact, any programming problem can be solved with a regular expression of the form

(a_1() *action*$_1$ | ... | a_n() *action*$_n$)*

where a_1, ..., a_n are all the input-event names. This approach has the advantage of obviously eliminating the possibility of unexpected events, but it does not take full advantage of the regular-expression paradigm, and amounts to little more than a roundabout way of associating a fixed event-handling routine with each input-event name.

Although the regular expression in Version 3 matches exactly the same set of input-event sequences as Version 2, Version 3 is somehow a more detailed model of the logical structure of the input stream for the purposes of this program. In effect, some of the state information captured in Version 2 by the use of variables is captured in Version 3 by the current position of the computation within the regular expression. This is demonstrated by our ability to annotate Version 3 with comments that reflect invariants that hold at certain points within the regular expression:

```
// Version 3′:

in {} Use, {} Inspection
out {} Violation
```

```
(   // no inspection needed
    Use()
    // inspection needed
    ( // inspection needed
      Use() { !>Violation({}); }
      // inspection needed
    )*
    Inspection()
    // no inspection needed
  |
    // no inspection needed
    Inspection()
    // no inspection needed
)*
```

Nonetheless, the original specification of the program was in terms of a pattern of events—two `Use` events without an intervening `Inspection` event—that should trigger a particular action. Many event-processing problems are specified in this way, and it is appealing to be able to model the event stream in a manner that directly reflects this specification. One way to do this is with an ambiguous set of alternatives such as

```
( . | Use() Use() {!>Violation({}); } )*
```

The wildcard alternative ensures that every input event matches the regular expression and the other alternative, in conjunction with the all-feasible-paths rule, ensures that if the most recently received events can be interpreted as matching the pattern `Use() Use()`, a `Violation` event will be emitted. Another way to model the event stream in a manner that directly reflects the specification is to use the regular-expression difference operator:

```
.* - ( .* Use() Use() { !>Violation({}); } )
```

The regular expression `.*` matches any sequence of input events, and the regular expression `.* Use() Use()` matches any sequence of input events ending with two `Use` events. Thus the difference regular expression as a whole matches any sequence of input events that does not end with two `Use` events. The key is that the action block `{ !>Violation(); }` is part of the second operand of the difference operator,[11] and, as Section 3.3.1 discussed, it is reached in the course of matching that operand.

4.2 Aggregating, smoothing, and debouncing

A common function in the processing of event streams is the aggregation or smoothing of data, particularly sensor data. Aggregation and smoothing are straightforward in EventScript. For example, the following aggregation program groups sensor readings into 20-reading bundles and emits the average reading for each bundle at the completion of that bundle:

[11] Because of the precedence of regular-expression operators, this would be the case even if the parentheses around the second operand were omitted. We include the parentheses to emphasize the crucial point that the action is part of the second operand.

```
in double Reading
out double Average

( { sum = 0.0; }
  ( Reading(x) { sum = sum + x; } ){20}
  { !>Average(sum/20); }
)*
```

Because the number *n* in an EventScript regular expression of the form *R*{*n*} must be a constant, the bundle size 20 is hardwired into the program. Here is a more general approach, in which the bundle size is specified by a `SetBundleSize` event preceding all the `Reading` events:

```
in long SetBundleSize, double Reading
out double Average

SetBundleSize(N)
{ sum = 0.0; seen = 0; }
( Reading(x)
  { sum = sum + x;
    seen = seen + 1;
    seen==N ? { !>Average(sum/N); sum = 0.0; seen = 0; }
  }
)*
```

The sliding-average example on page 9 smoothes a data stream by emitting, for each data reading received, the average of the *N* most recent readings, where *N* is specified by a `SetWindowSize` event preceding all readings. An exponential moving average is even easier to compute. The following program uses a smoothing factor alpha specified by a `SetSmoothingFactor` event preceding all readings:

```
in double SetSmoothingFactor, double Raw

out double Smoothed

SetSmoothingFactor(alpha) { oneMinusAlpha = 1.0 - alpha; }
Raw(history)
( { !>Smoothed(history); }
  Raw(x)
  { history = alpha*x + oneMinusAlpha*history ; }
)*
```

Debouncing is a discrete analog to smoothing. It entails aggregating several input signals resulting from a single real-world occurrence into a single output signal. For example, when a key on a keyboard is pressed, electrical contacts may “bounce” together and apart several times (or electrical arcing may occur between two contacts as the distance between them approaches zero), resulting in several signals that ought to be interpreted as a single key press. The following EventScript program interprets each incoming signal from a given key as attributable to the same key press as the previous signal from that key, unless five milliseconds have passed since the previous signal from that key:

```
in long Signal group(Signal)
out long KeyPress

( Signal(k) { !>KeyPress(k); } Signal()* elapse[5 msec]() )*
```

(An input signal from a given key is manifested as a `Signal` input event carrying a number that identifies the key. For each distinct key press it detects, the program emits a `KeyPress` output event carrying the same number. Signal events are grouped by key numbers, so that separate instance of this computation runs for each key on the keyboard. Although this computation is nonterminating, the number of keys on the keyboard is fixed, resource leaks of the kind discussed in Section 3.2 are not a concern.) Essentially the same logic is found in a program that collects RFID readings at a toll booth, and treats a reading with a given tag ID as attributable to the same passage through the toll booth as the previous RFID reading with that tag ID, unless five minutes have passed since the previous reading with that tag ID:

```
in {string readerID; string tagID; time when;} Reading
      group(Reading.tagID)

out {string readerID; string tagID; time when;} Passage

Reading(r) { !>Passage(r); }
Reading()*
elapse[5 minutes]()
```

(In this case, input events are grouped by tag ID, and resource leaks *are* a concern. Therefore, a computation instance terminates when five minutes have elapsed since the last reading for its tag.)

In some applications, the possible values of a continuous variable measuring a physical property are summarized by a collection of discrete values, and we continually receive input events, reporting either the continuous variable itself or the corresponding discrete state. Suppose the value of the continuous variable is near the boundary between regions corresponding to two discrete states *A* and *B*. As the value of the continuous variable varies in a noisy manner, but in the general direction away from the *A* region and towards the *B* region, we are likely to receive signals reflecting the discrete model of the system settled in state *A*, then oscillating chaotically between states *A* and *B*, and finally settled in state *B*. The role of debouncing in such a case is to aggregate these input signals into an output signal reporting one clean transition, without oscillation, from state *A* to state *B*.

Typical debouncing strategies in such cases are sticky, continuing to report the old discrete state until faced with convincing evidence that the system is in the new discrete state. For debouncing logic that receives reports of the continuous variable itself, this convincing evidence might be a report that the continuous variable has crossed a certain distance past the boundary between regions corresponding to different discrete states (*distance* debouncing). For debouncing logic that receives reports of the discrete state corresponding to the current value of the continuous variable, this convincing evidence might be the receipt of *n* consecutive signals that the system is in the new state, for some value of *n* (*count* debouncing), or the receipt only of signals that the system is in the new state for some time *t* (*time* debouncing).

The following program receives reports of a continuous variable—temperature in a refrigeration system—and performs distance debouncing, indicating that the system has switched to either a cold state (below 40 degrees) or a warm state (40 degrees or higher) when the temperature has crossed 5 degrees past the boundary between the new state and the old state. To err on the side of reporting near-borderline temperatures as warm, we assume that the system starts out in the warm state.

```
  in double Temperature
      case {   Temperature < 35.0 ? DefinitelyCold
             : Temperature < 40.0 ? SlightlyCold
             : Temperature < 45.0 ? SlightlyWarm
             : DefinitelyWarm
      }

  out {} Warm, {} Cold

  { !>Warm({}); } // initial report
  ( // Warm state
    (.-DefinitelyCold())* DefinitelyCold()
    { !>Cold({}); }
    // Cold state
    (.-DefinitelyWarm())* DefinitelyWarm()
    { !>Warm({}); }
    // Warm state
)*
```

The following EventScript program performs count debouncing for n=5, aggregating `RawOff` and `RawOn` input signals into `DebouncedOff` and `DebouncedOn` output signals:

```
in { } RawOff, { } RawOn

out { } DebouncedOff, { } DebouncedOn

(  // Off state
   ( RawOn(){0,4} RawOff() )*
   RawOn(){5}
   { !>DebouncedOn({}); }
   // On state
   ( RawOff(){0,4} RawOff() )*
   RawOff(){5}
   { !>DebouncedOff({}); }
   // Off state
)*
```

In the *Off* state, we expect a sequence of zero or more `RawOff` events, each potentially preceded by up to four `RawOn` events; but when we see five consecutive `RawOn` events, we transition to the *On* state. The *On* state works analogously.[12]

The following EventScript program works with the same input and output signals, but performs time debouncing for t=0.5 seconds:

[12] This regular expression is ambiguous. For example, in the *Off* state, a sequence 1, 2, 3, or 4 consecutive `RawOn` events may correspond to either `RawOn(){0,4}` or `RawOn(){5}`. However, once a `RawOff` event or a fifth `RawOn` event arrives, only one of these paths remains feasible. Since there is only one feasible path at each action, the ambiguity is innocuous.

```
in { } RawOff, { } RawOn

out { } DebouncedOff, { } DebouncedOn

 ( // Off state
   ( RawOff()* RawOn()+ RawOff() )*
   RawOn()
   { deadline = millisecondsAfter(500, now() ); }
   RawOn()*
   arrive[(deadline)]()
   { !>DebouncedOn({}); }
   // On state
   ( RawOn()* RawOff()+ RawOn() )*
   RawOff()
   { deadline = millisecondsAfter(500, now() ); }
   RawOff()*
   arrive[(deadline)]()
   { !>DebouncedOff({}); }
   // Off state
)*
```

The regular expression `( RawOff()* RawOn()+ RawOff() )*` matches what we may call a *false-start sequence*, consisting of zero or more `RawOff` events, one or more `RawOn` events, and then a `RawOff` event (occurring less than 500 milliseconds after the first `RawOn` event of the sequence). In the *Off* state, we expect zero or more false-start sequences. Eventually, we may see a `RawOn` event that will not be followed by any `RawOff` event for the next 500 milliseconds. This `RawOn` event occurrence *e* is followed by zero or more `RawOn` events, and then the arrival of the time 500 milliseconds after *e*, at which point we enter the *On* state. The *On* state works analogously.[13]

4.3 Reacting to the absence of an event

A common task in event processing is to react to the absence of an event in a certain time interval whose beginning and end are marked by delimiting events. The delimiting event marking the end of the interval may be an ordinary named input event, the arrival of a particular time and date, or the passage of a certain amount of time since the beginning of the interval. Suppose we are expecting a repeated series of intervals delimited by `Start` and `Stop` events, each containing zero or more `Signal` events, and we are to emit a `NoSignal` event at the end of each interval that did not contain a `Signal` event. Here is a straightforward solution:

[13] This regular expression is also ambiguous. After the program has entered the Off state and executed zero or more false-start sequences, an arrive `RawOn` event potentially matches both `RawOn()+` in the false-start sequence and the event marker `RawOn()` appearing just before the assignment to `deadline`. Since any action reached along any feasible path is executed, `deadline` is set at this point to the time 500 milliseconds after the current time. Subsequent `RawOn` events arriving at this point match both `RawOn()+` in the false-start sequence and `RawOn()*` just after the assignment to `deadline`. If the next event is the arrival of the time `deadline`, the only remaining feasible path is the one that brings us to the emit action for `DebouncedOn`; but if the next event is a `RawOff` event, the only feasible path is the one inside the false-start sequence.

```
in {} Start, {} Signal, {} Stop
out {} NoSignal

( Start()
  { signalReceived = false; }
  ( Signal() { signalReceived = true; } )*
  Stop() { !signalReceived ? !>NoSignal({}); }
)*
```

It can be argued that the following solution, which does not involve any variables, is even simpler:

```
in {} Start, {} Signal, {} Stop
out {} NoSignal

( Start()
  ( Stop() { !>NoSignal({}); } | Signal()+ Stop() )
)*
```

Here is a variation in which the interval extends from 9:00 a.m. to 5:00 p.m. every day:

```
in {} Start, {} Signal, {} Stop
out {} NoSignal

( arrive[9:00]()
(   arrive[17:00]() { !>NoSignal({}); }
  | Signal()+ arrive[17:00]()
  )
)*
```

Here is a variation in which the interval starts upon the arrival of a `Start` event and ends one hour later:

```
in {} Start, {} Signal, {} Stop
out {} NoSignal

( Start() { endTime = hoursAfter(1, now()); }
  (   arrive[(endTime)]() { !>NoSignal({}); }
    | Signal()+ arrive[(endTime)]()
  )
)*
```

One special case of the absence of an event is a time-out. The following program echoes[14] `Reading` events, but also emits a `MissingReading` event each time 15 seconds elapse without an input event:

```
in double Reading
out double Reading, { } MissingReading

(   Reading(x) { !>Reading(x); }
  | elapse[15 seconds]() { !> MissingReading({}); }
)*
```

[14] As this example illustrates, EventScript allows a program to use a single name as both an input-event name and an output-event name, but in many contexts this practice may be confusing to readers of the program.

This program is similar to the first program on page 7, which reports the average of three sensors every time a new reading arrives from some sensor and every time ten seconds have elapsed since an input arrived from any sensor. The second program on page 7, which reports the average of three sensors at regular ten-second intervals, illustrates the use of time-outs for periodic processing. The keyboard debouncing example on page 26 and the toll-booth debouncing example on page 27 illustrate yet another use of time-outs.

4.4 Factoring alternatives

Sometimes a set of alternatives—say

$$a_1 \mid \ldots \mid a_j \mid a_{j+1} \mid \ldots \mid a_k \mid a_{k+1} \mid \ldots \mid a_n$$

—includes identical actions in several alternatives. In such cases, we can avoid redundant coding by grouping alternatives with actions in common into a single alternative, as in $(a_1 \mid \ldots \mid a_j) \mid (a_{j+1} \mid \ldots \mid a_k) \mid (a_{k+1} \mid \ldots \mid a_n)$. The actions common to all the alternatives in a group can then be placed after that group. The resulting finite-state machine is identical, but the resulting program is easier to maintain because any change that must be made to the common actions can be made in a single place.

Consider a one-floor industrial plant that uses an active-badge-based location-tracking system to enforce a policy that all visitors must be accompanied by their hosts at all times. The location-tracking system uses *x* and *y* coordinates in which one unit corresponds to a distance of one meter. We assume that visitors and hosts meet at a security desk near the entrance to the plant, with coordinates x=3.5, y=7.2. At this desk, a security officer assigns a visitor ID to the visitor and records the fact that a particular host is responsible for a particular visitor. One host may be responsible for several visitors. Every time the active badge of a visitor is detected, the location-tracking system generates a `VisitorLocation` event containing the visitor ID and *x* and *y* coordinates of the visitor. Every time the active badge of a host is detected, the location-tracking system generates a `HostLocation` event for each visitor assigned to that host, containing the ID of the *visitor* and the *x* and *y* coordinates of the *host*. When the visitor leaves the building, the security officer records this fact, and the system emits an `Unregister` event containing the visitor ID of the departing visitor. We are to generate an `UnaccompaniedVisitor` event, containing a visitor ID and the last known coordinates of the visitor, any time a visitor's last reported position is more than 10 meters away from the host's last reported position, or any time no location has been reported for one of these individuals for one minute.

Our EventScript program groups `VisitorLocation`, `HostLocation`, and `Unregister` events by visitor ID, so that one instance of the program runs for each currently registered visitor; the `Unregister` event allows us to write a terminating program, so that there is no resource leak. (See Section 3.2.) The instance corresponding to a given visitor ID saves the time by which we expect to receive the next `VisitorLocation` event for that visitor ID in the variable `visitorDeadline`, and the time by which we expect to receive the next `HostLocation` event for that visitor ID in the variable `hostDeadline`.

Here is an initial version of the program:

```
in { string visitorID; double x; double y; }
      HostLocation group(HostLocation.visitorID),
   { string visitorID; double x; double y; }
      VisitorLocation group(VisitorLocation.visitorID),
   string Unregister group(Unregister)

out { string visitorID; double x; double y; }
       UnaccompaniedVisitor

{ THRESHOLD = 10;
  THRESHOLD_SQUARED = THRESHOLD*THRESHOLD;
  INITIAL_LOCATION = {visitorID: group[0], x: 3.5, y: 7.2};
  h = INITIAL_LOCATION;
  v = INITIAL_LOCATION;
  hostDeadline = minutesAfter(1, now());
  visitorDeadline = hostDeadline;
}

(   HostLocation(h)
    { hostDeadline = minutesAfter(1, now());
      deltaX = v.x-h.x;
      deltaY = v.y-h.y;
      deltaX*deltaX + deltaY*deltaY > THRESHOLD_SQUARED ?
         !>UnaccompaniedVisitor
              ({visitorID: group[0], x: v.x, y: v.y});
      }
  |
    VisitorLocation(v)
    { visitorDeadline = minutesAfter(1, now());
      deltaX = v.x-h.x;
      deltaY = v.y-h.y;
      deltaX*deltaX + deltaY*deltaY > THRESHOLD_SQUARED ?
         !>UnaccompaniedVisitor
              ({visitorID: group[0], x: v.x, y: v.y});
    }
 |
   arrive[(hostDeadline)]()
   { hostDeadline = minutesAfter(1, hostDeadline);
     !>UnaccompaniedVisitor
          ({visitorID: group[0], x: v.x, y: v.y});
   }
 |
   arrive[(visitorDeadline)]()
   { visitorDeadline = minutesAfter(1, visitorDeadline);
     !>UnaccompaniedVisitor
          ({visitorID: group[0], x: v.x, y: v.y});
   }
)*
Unregister()
```

Except for the resetting of either `hostDeadline` or `visitorDeadline`, the actions performed after the arrival of a `HostLocation` event are identical to those performed after the arrival of a `VisitorLocation` event, and the actions performed after the arrival of the time `hostDeadline` are identical those performed after the arrival of

the time `visitorDeadline`. Therefore, the repetition regular expression in this program can be factored as follows:

```
(
    (   HostLocation(h)
        { hostDeadline = minutesAfter(1, now()); }
      |
        VisitorLocation(v)
        { visitorDeadline = minutesAfter(1, now()); }
    )
    { deltaX = v.x-h.x;
      deltaY = v.y-h.y;
      deltaX*deltaX + deltaY*deltaY > THRESHOLD_SQUARED ?
         !>UnaccompaniedVisitor
              ({visitorID: group[0], x: v.x, y: v.y});
    }

  |

    (   arrive[(hostDeadline)]()
        { hostDeadline = minutesAfter(1, hostDeadline); }
      |
        arrive[(visitorDeadline)]()
        { visitorDeadline = minutesAfter(1,visitorDeadline); }
    )
    { !>UnaccompaniedVisitor
          ({visitorID: group[0], x: v.x, y: v.y}); }

)*
```

4.5 Pipelining EventScript programs

Many event-processing problems can be simplified by decomposing them into stages of a pipeline, in which the output events emitted by one stage are fed as input events into the next stage. Sometimes the simplification results from separating different aspects of the problem into different stages, or by filtering out irrelevant information. Sometimes the simplification results from resolving a clash between the structure of the original input stream and the structure of the ultimate output stream by introducing an intermediate stream whose structure is compatible with both. Sometimes the simplification results from the use of event grouping to group data in different ways at different stages.

Suppose a piece of equipment issues status reports every few minutes, reporting either normal or abnormal status. A *window* begins every hour on the hour. When we observe three consecutive windows containing at least one abnormal reading, we issue an alarm and reset the count of consecutive abnormal windows to zero. The alarm should be issued as soon as the first abnormal status report of the third consecu-

tive abnormal window is received. The following one-stage EventScript program solves this problem:[15]

```
in boolean Status
   case { Status ? NormalStatus : AbnormalStatus }

out { } Alarm

 ( NormalStatus()*                // (0,N) -> (0,N)
   (
       arrive[.:00]()             // (0,N) -> (0,N)
     |
       AbnormalStatus()           // (0,N) -> (0,A)
       .*                         // (0,A) -> (0,A)
       arrive[.:00]()             // (0,A) -> (1,N)
       NormalStatus()*            // (1,N) -> (1,N)
       (    arrive[.:00]()        // (1,N) -> (0,N)
         |
            AbnormalStatus()      // (1,N) -> (1,A)
            .*                    // (1,A) -> (1,A)
            arrive[.:00]()        // (1,A) -> (2,N)
            NormalStatus()*       // (2,N) -> (2,N)
            (   arrive[.:00]()    // (2,N) -> (0,N)
              |
                AbnormalStatus() // (2,N) -> (2,A)
                { !>Alarm({}); } // (2,A) -> (2,A)
                .*                // (2,A) -> (2,A)
                arrive[.:00]()    // (2,A) -> (0,N)
            )
       )
   )
)*
```

---

[15] The invariant holding at each point in this program can be described by a pair (*k*,*status*), where *k* is an integer in the range 0 to 2 and *status* is either "N" (for *normal*) or "A" (for *abnormal*). The invariant (*k*,*status*) asserts that the immediately preceding *k* windows are to be counted as abnormal, and that at least one abnormal status report has (if *status*="A") or has not (if *status*="N") been seen in the current window. The comments of the form `// (`k_1`,`$status_1$`) -> (`k_2`,` $status_2$`)` at the ends of the lines in this program assert that the invariant (k_1, $status_1$) holds before the line and the invariant (k_2, $status_2$) holds afterward. The validity of these comments can be derived from the following rules of inference:

- Whatever invariant holds before `NormalStatus()` still holds afterward.
- If (*k*,N) holds before `AbnormalStatus()`, then (*k*,A) holds afterward.
- If (*k*,A) holds before `AbnormalStatus()`, then (*k*,A) still holds afterward. (Since the wild-card `.` is equivalent to `NormalStatus() | AbnormalStatus()`, it follows that if (*k*,A) holds before `.`, then (*k*,A) holds afterward.)
- For any regular expression *R* that preserves a given invariant, the regular expression *R** also preserves that invariant.
- If (*k*,N) holds before `arrive[.:00]()`, then (0,N) holds afterward (because a window with no abnormal status reports has just ended).
- If (*k*,A) holds before `arrive[.:00]()`, then (*k*+1,N) holds afterward (because an abnormal window has just ended and no status reports have arrived in the current window).
- Whatever invariant holds before `{ !>Alarm({}); }` still holds afterward.

However, a simpler and more obviously correct solution can be obtained by decomposing the processing into two stages. The first stage consumes `NormalStatus`, `AbnormalStatus`, and `arrive[.:00]()` events, and emits either a `NormalWindow` event or an `AbnormalWindow` event at the end of each window. The second stage consumes `NormalWindow` and `AbnormalWindow` events and emits `Alarm` events after three consecutive `AbnormalWindow` events for which no `Alarm` event was previously emitted. Here is the first stage:

```
in boolean Status
   case { Status ? NormalStatus : AbnormalStatus }

out { } NormalWindow, { } AbnormalWindow

(
  NormalStatus()*
  (   arrive[.:00]() { !>NormalWindow({}); }
    |
      AbnormalStatus() { !>AbnormalWindow({}); }
      .*
      arrive[.:00]()
  )
)*
```

Here is the second stage:

```
in { } NormalWindow, { } AbnormalWindow
out { } Alarm

(
  ( AbnormalWindow(){0,2} NormalWindow() )*
  AbnormalWindow(){3} { !>Alarm({}); }
)*
```

We have separated concern with determining when a window has ended and whether the window was abnormal from concern about whether there have been three consecutive abnormal windows. Furthermore, instead of dealing with an event for each status report, the second stage only has to deal with one summary event for each window.

The three-big-trades example on page 12 can also be improved by decomposing it into two stages. The first stage consumes `BigTrade` and `SmallTrade` events, filters out the `SmallTrade` events, and emits a `FilteredTrade` event corresponding to each `BigTrade` event:

```
in { string traderID;
     string company;
     double price;
     long volume;
     time when;
   } Trade
        case { Trade.price*Trade.volume>=10000.0 ?
                    BigTrade
                 : SmallTrade
             }
```

```
out
   {string traderID; string company; double value; time when;}
      FilteredTrade

(   SmallTrade()
  |
    BigTrade(t)
    { !>FilteredTrade
           ( { traderID: t.traderID,
               company: t.company,
               value: t.price*t.volume,
               when: t.when }
            );
    }
)*
```

The second stage consumes `FilteredTrade` events and emits `ThreeBigTrade` events:

```
in {string traderID; string company; double value; time
when;}
      FilteredTrade
      group(FilteredTrade.traderID, FilteredTrade.company)

out {string traderID; string company; double averageValue;}
       ThreeBigTrades

// 0 trades pending
FilteredTrade(t1) { expiration1 = hoursAfter(1,t1.when); }
// 1 trade pending
(
   FilteredTrade(t2) { expiration2=hoursAfter(1,t2.when); }
   // 2 trades pending
   (  FilteredTrade(t3)
      { outputValue =
          { traderID: t3.traderID,
            company: t3.company,
            averageValue: (t1.value+t2.value+t3.value)/3 };
        !>ThreeBigTrades(outputValue);
        t1 = t2;
        t2 = t3;
        expiration1 = expiration2;
        expiration2 = hoursAfter(1,t2.when);
      }
   )*
   arrive[(expiration1)]()
   { expiration1 = expiration2; t1 = t2; }
   // 1 trade pending
)*
arrive[(expiration1)]()
// 0 trades pending; terminate
```

The second stage of the revised program is a simplified version of the body of the repetition in the original program, in which small trades are not addressed and in which the values of trades (which are computed in the first stage) are taken from

`FilteredTrade` input events rather than computed. The event grouping in the original program is confined to the second stage of the modified program. This has important performance implications: The original program maintained a computation instance for every combination of trader ID and company, but the second stage of the modified program maintains a computation instance only for those trader-company combinations for which there are big trades. Since we can expect the overwhelming majority of traders to execute only small trades, the modified program requires far fewer computation instances. Still, in accordance with the discussion in Section 3.2 about resource leaks, we have transformed the second stage of the modified program from a nonterminating program into a terminating one, by replacing the outermost repetition with the body of that repetition. In the revised program, a computation instance of the second stage terminates when the most recent big trade for its trader-company combination has become an hour old, so that the resources associated with that combination can be freed. (Should there later be another big trade by that trader for that company, a new computation instance will be created.)

A multistage solution is also helpful to reconcile what M.A. Jackson [Jac75] calls a *boundary clash*. Suppose a sensor takes readings once every second and a controller attached to the sensor packs consecutive readings into a message containing 15 consecutive readings in order, delivered at 15-second intervals. Suppose further that we are to emit an event three times a minute, each containing the largest of the most recent 20 readings. The boundary clash is between an input stream in which events correspond to groups of 15 readings and an output stream in which events correspond to groups of 20 readings. We solve the problem by introducing an intermediate stream in which events correspond to groups of five readings. The first stage decomposes a 15-reading input event into three five-reading output events. The second stage processes four five-reading input events at a time to produce an output event. We assume that sensor readings are nonnegative real numbers. Here is the first stage:

```
in double[] PacketOf15
out double[] PacketOf5

{ buffer = new double[5]; }
(
   PacketOf15(input)
   { for (i: 0, 3) {
        for (j: 0, 5) buffer[j] = input[3*i+j];
        !>PacketOf5(buffer);
     }
   }
)*
```

Here is the second stage:

```
in double[] PacketOf5
out double MaxOf20

(
   { max = 0.0; }
   ( PacketOf5(input)
     { for (i: 0, 4) { input[i] > max ? max = input[i]; } }
   ){4}
   { !>MaxOf20(max); }
)*
```

Jackson solves such problems by writing two coroutines, one of which feeds values to the other, and then performing intricate program transformations to implement the coroutines in a conventional programming language. In the EventScript solution, the two stages of the pipeline play the role of these coroutines.

Our final example of pipelining illustrates the power of grouping events by different criteria in different stages of the pipeline. This example uses radiation sensors to monitor the current radiation level in several rooms of an industrial plant, and uses an active-badge-based location-tracking system to detect the presence of specific employees in specific rooms, to ensure that no employee receives a dose of 1.0 milliSievert[16] or more during any 24-hour period. Each minute during which an employee's exposure for the most recent 24 hours (1,440 minutes) equals or exceeds 1.0 milliSievert, we are to issue an alert specifying the employee and the total exposure over the past 24 hours. The radiation sensor for a room emits `RadiationLevel` events whose values identify the room and its current radiation level in milliSieverts per hour. Every time it reads an active badge, the location-tracking system emits an `EmployeeLocation` event whose value reports that a particular employee has been detected in a particular room. The role of the first stage is to join the `RadiationLevel` data to the `EmployeeLocation` data, and report that particular employees have been detected at places with particular radiation levels. The first stage is grouped by room, so that all `RadiationLevel` events and all `EmployeeLocation` events for a given room go to the corresponding instance of the first-stage computation. When an instance of the first stage receives a `RadiationLevel` event for its room, it simply saves that event's value. When it receives an `EmployeeLocation` event reporting that a particular employee has been detected in that room, it emits an `EmployeeExposure` event, reporting that that employee has been detected in a place whose radiation level is the most recently saved radiation level for the room:

```
in { string roomID; double mSvPerHour; } RadiationLevel
      group(RadiationLevel.roomID),
   { string employeeID; string roomID; } EmployeeLocation
      group(EmployeeLocation.roomID)

out { string employeeID; double mSvPerHour; } EmployeeExposure

{ r = {roomID: "", mSvPerHour: 0.0 }; }
(
      RadiationLevel(r)
   |
      EmployeeLocation(e)
      { !>EmployeeExposure
           ( { employeeID: e.employeeID,
               mSvPerHour: r.mSvPerHour }
           );
      }
)*
```

(The value of the most recently received `RadiationLevel` event is stored in the variable `r`. This variable is initialized to a dummy event value with a radiation level of zero, to be used for `EmployeeLocation` events that arrive before the first `Radia-`

[16] 1.0 milliSievert = 100 milliRem.

tionLevel event.) The various instances of the first stage emit their EmployeeExposure events into one merged stream. The role of the second stage is to track the total exposure of individual employees for the past 24 hours. The second stage is grouped by employee, so all the EmployeeExposure events for a given employee are handled by the same instance of the computation. The regular expression for the second stage matches repeated sequences each consisting of zero or more EmployeeExposure events followed by the arrival of the beginning of a new minute. Each minute, we compute the average *r* of the radiation levels in EmployeeExposure events that arrived for that employee during the previous minute, and we make the approximating assumption that the employee was exposed to a radiation level of *r* milliSieverts per hour (or *r*/60 milliSieverts per minute) uniformly for the entire minute, so that the employee's total exposure for the minute is *r*/60 milliSieverts. Each instance has a 1,440-element array used as a circular buffer to store the exposure computed for each minute in the last 24 hours. (24 hours = 1,440 minutes.) Rather than sum the elements of the buffer each minute, we maintain a running sum, as in the sliding-average example on page 9. Each minute, we remove the oldest value from the buffer, subtract it from the running sum, add the exposure value for the most recent minute to the running sum, and place that value in the buffer. Here is the second stage:

```
in
   { string employeeID; double mSvPerHour; }
      EmployeeExposure
   group(EmployeeExposure.employeeID)

out { string employeeID; double mSv; } ExposureLimit

{ THRESHOLD = 1.0; MINUTES_PER_DAY = 24*60;
  history = new double[MINUTES_PER_DAY];
  for (i: 0, MINUTES_PER_DAY-1) history[i] = 0.0;
  mSvInLast24Hours = 0.0; circularBufferCursor = 0;
}

(
   { readingTotalForMinute = 0.0;
     readingCountForMinute = 0;
   }
   (
      EmployeeExposure(report)
      { readingTotalForMinute =
           readingTotalForMinute+report.mSvPerHour;
        readingCountForMinute = readingCountForMinute+1;
      }
   )*
   arrive[.:.:00]()
```

```
    {
       mSvPerHourAverage =
          readingCountForMinute==0 ?
               0.0
             : readingTotalForMinute/readingCountForMinute;
       mSvPerMinuteAverage = mSvPerHourAverage/60.0;
       mSvInLast24Hours =
            mSvInLast24Hours
          - history[circularBufferCursor]
          + mSvPerMinuteAverage;
       history[circularBufferCursor] = mSvPerMinuteAverage;
       circularBufferCursor =
          (circularBufferCursor+1) % MINUTES_PER_DAY;
       mSvInLast24Hours >= THRESHOLD ?
          !>ExposureLimit
             ({employeeID: group[0], mSv: mSvInLast24Hours});
    }
)*
```

Both stages of this pipeline are nonterminating, but we expect the number of rooms to remain fixed and the number of employees to be relatively stable, so the possibility of a resource leak is not an issue.

5. Connecting EventScript to the outside world

EventScript is based on a sparing model of an event object, compatible with a wide variety of real-world event messages and imposing minimal run-time overhead. We also aim to isolate the semantics of EventScript from the notion of event channels in the outside world, to shield the semantics of EventScript from unnecessary complication and to make EventScript widely applicable.

An EventScript run-time implementation includes pluggable components called *adapters*. An input adapter is responsible for somehow collecting event signals from the outside world, translating them into EventScript event objects, and feeding them to the EventScript engine. An output adapter is responsible for taking the event objects emitted by the EventScript engine, translating them into some external format, and somehow dispatching them to the outside world.

The event objects emitted by the EventScript engine include the error events emitted, as described in Section 3.1, in response to runtime errors. It is the output adapter that is responsible for handling error events. This may entail, for example, ignoring the error event, displaying its message on an operator's console, logging it, or transmitting it to some downstream event consumer as if it were an ordinary event.

Event objects are matched against regular expressions based on their event names. The name of an event object also determines the data type of the event's value. However, there are several ways to define what it means for an event entering the EventScript run-time environment from the outside world to be translated into an event object with a particular event name:

- The name of an event object indicates the event channel on which it arrived. (All event objects originating from the same event channel must have values of

the same data type, and event objects originating from different event channels must have different names, even if they have values of the same data type.)

- The name of an EventScript event object corresponds to some sort of event name in the outside world. (All events with the same outside-world event name must be translated into EventScript event objects with values of the same data type.)
- The name of an event object is simply a tag indicating the data type of the event's value. (Event objects whose values have identical data types may have the same event name even if they arrive on different event channels.)

The definition of EventScript does not impose any of these choices, since there are scenarios in which any one of them might be appropriate, depending on the nature of the outside sources of events. Adapters implementing any of these choices are consistent with EventScript. Indeed, the semantics of EventScript are agnostic about the existence of multiple external event channels: The role of an input adapter is to collect event notifications, from whatever source, and feed corresponding event objects to the EventScript engine one at a time, in some sequence. EventScript regular expressions are matched against this totally ordered sequence.

The EventScript language makes no assumptions about the external representation of event objects. It is the responsibility of an adapter to convert, according to its own conventions, between the EventScript event-object model and the event representations of the outside world. For example, an input adapter might derive the event type of an input event from an event name that is part of the incoming event notification, or from a particular attribute found in every incoming event notification, or from a computation based on the form and content of the incoming event notification.

The rules of the language allow, but do not depend upon, some correspondence between EventScript event names and notions of message types in the outside world. For example, in the outside world there might be an inheritance-based message-type system, with a message type `PresenceStatus` having a `userid` field and an `availability` field, and a message subtype `RichPresenceStatus` extending `PresenceStatus` with a `location` field and a `preferredCommunicationMode` field. EventScript would treat `PresenceStatus` and `RichPresenceStatus` event objects with distinct names, whose values belong to two independent event types, the first with `userid` and `availability` structure fields and the second with `userid`, `availability`, `location`, and `preferredCommunicationMode` structure fields. A regular expression matching an event named `PresenceStatus` would not match an event named `RichPresenceStatus`, but the EventScript programmer would be able to write a regular expression such as

```
(   PresenceStatus(ps) { u = ps.userid; }
  | RichPresenceStatus(rps) { u = rps.userid; }
)
{ inList(buddies, u) ?
    {   a=="available" ? !>MarkAvailable(u);
      : a=="away"      ? !>MarkAway(u);
      : a=="offline"   ? !>MarkOffline(u);
    }
}
```

that matches an event with either name and responds to the two kinds of events in the same manner.

6. EventScript development tools and execution platforms

Until now, we have discussed the EventScript language, independent of any particular implementation. This section is concerned with one particular implementation of the language. We have developed a run-time engine that is easily plugged into various execution environments, as well as a number of tools for developing EventScript programs. The tools include a language-sensitive editor, a compiler, and a tester that runs EventScript programs against a timestamped event trace. We have also defined and implemented a Java API for writing input and output adapters and externally defined functions.

6.1 EventScript engine

The EventScript engine loads an XML file that contains a representation of a finite-state machine, then runs the finite-state machine. The heart of the XML file is a transition table specifying, for each state and input event[17], an action to be performed and a new state. The XML file also contains information about grouping keys, event classification, externally defined functions, the initial state, and actions to be performed before the arrival of the first input event.

An arriving input event is represented by an object that includes the value of the event. In the case of a named event, the object also includes the event name. In the case of a time event, the object also includes the identity of the corresponding event-marker occurrence in the source text and, in the case of a program with event grouping, the grouping-key value for the computation instance that triggered the time event.

In a program without event grouping, the execution of the state machine consists of performing any initial actions, and then responding to each input event by looking up the transition-table entry corresponding to the current state and the arriving event, performing the actions found in that entry, and updating the current state to the state found in that entry. Among the transition-table entries for the current state, the table entry corresponding to the arriving event is selected as follows: If the event has a name declared in an event declaration with an event-case clause, the conditions of the event-case clause are evaluated in turn until the applicable arm is found, and the event-name in that arm is used for the table lookup; for any other named event, the event name itself is used for the table lookup; for a time event, the identity of the corresponding event-marker occurrence is used for the table lookup.

[17] For each state, the transition table contains an entry for each *symbol* of the state machine's *input alphabet*. The input alphabet consists of a distinct symbol for each of the following:

- each input event name declared in an event declaration without a case clause
- each classified-event name appearing in an arm of a case clause
- each textual occurrence of a time event marker

In a program with event grouping, the EventScript engine maintains a hash table that maps grouping-key values to *instance-context* objects. Each instance-context object holds the current finite-automaton state and variable values for the computation instance corresponding to a different grouping-key value. When an input event arrives, its grouping-key value is computed, and the hash table is probed to see if there is an existing instance-context object for that value. If not, a new instance-context object is created in the initial finite-automaton state and any initial actions are executed using the variables in that instance-context object. In either case, we now have an instance-context object corresponding to the grouping-key value. The event is now processed in the same manner is an a program without event grouping, but with actions reading and updating only the variable instances in that instance-context object. If the EventScript program terminates (i.e., reaches a finite-automaton state marked as being a final state), the instance-context object is removed from the hash table and its storage is released. (Section 3.2 noted the possibility of a resource leak in nonterminating programs with event grouping. This resource leak manifests itself in the unbounded accumulation of instance-context objects in the hash table.)

In addition to actions explicitly appearing in the source program, the initial actions and the actions in the transition table may include *timer start actions* and *timer stop actions*. A timer start action is executed upon entry to a state in which the next event may be a time event; this execution entails evaluating the expressions in `arrive` and `elapse` event markers reachable from that state, determining the lexically earliest time-event marker among those with the earliest execution time,[18] and setting a timer to generate a time event corresponding to that marker when that time arrives. A timer stop action is executed upon departure from a state in which the next event may be a time event, to cancel the pending time event. For example, Fig. 6 shows the state diagram, including timer start and timer stop actions, for the following program:

```
in {} CallReceived
out long BusinessHourCallCount

( CallReceived()*
   arrive[9:00]() { count = 0; }
   ( CallReceived() { count = count + 1; } )*
   arrive[17:00]() { !>BusinessHourCallCount(count); }
)*
```

[18] In the case of an `arrive` event marker with wildcards, this entails determining the next date and time that will match the date-time pattern in the event marker.

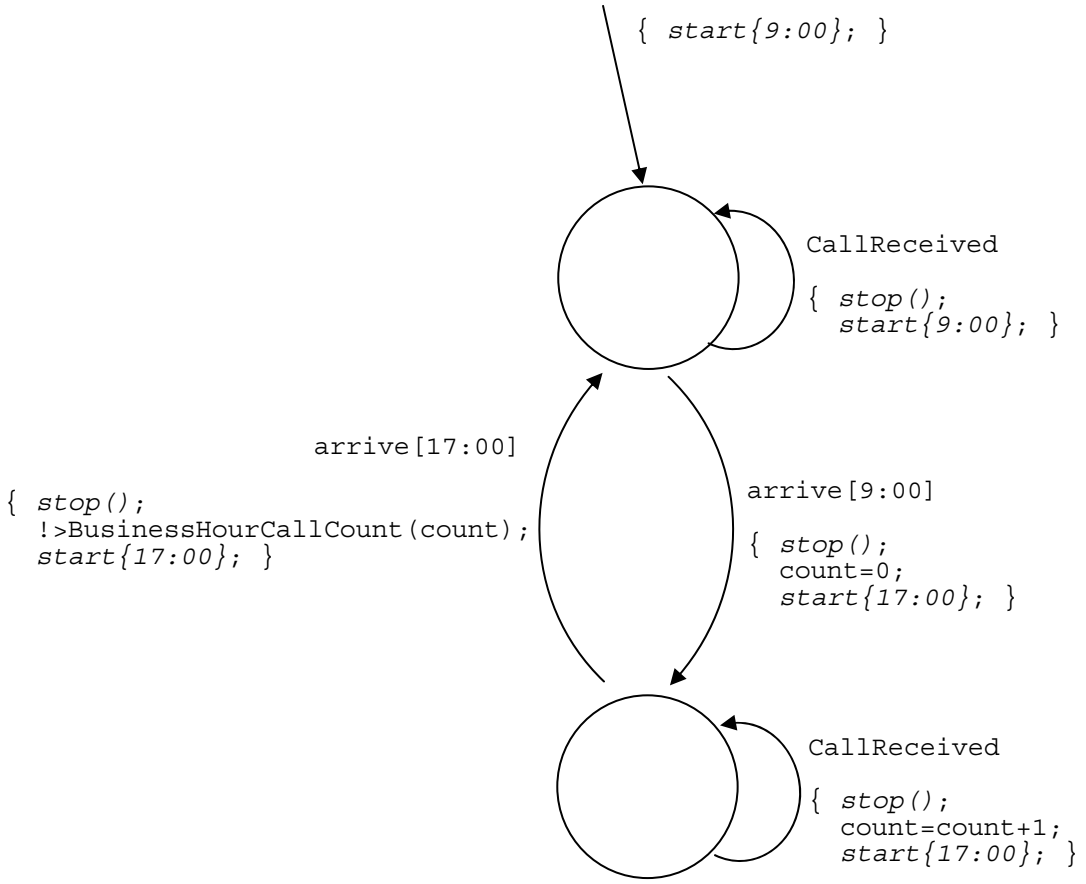

Fig. 6. A state-transition diagram in which each transition is labeled with the event causing the transition to be taken and the actions, including timer start and timer stop actions, performed when that transition is taken. The arrow entering the top state is labeled with the initial action to be executed before any event arrives. The action `start(pattern)` sets a timer to trigger a time event at the next date and time matching `pattern`, and the action `stop()` turns the timer off, canceling any pending trigger.

As illustrated in Fig. 7, our Java implementation of the EventScript run-time machinery consists of the EventScript engine plus a number of pluggable components that connect the EventScript engine to the surrounding environment. The input adapter learns of events in the outside world, builds corresponding EventScript event objects, and feeds the event objects to the EventScript engine by calling a method that takes an event object as a parameter. When the EventScript engine emits an event, it calls an analogous method of the output adapter, which receives an event object for the emitted event as a parameter and takes appropriate actions in the outside world in response to the emitted event. (Input and output adapters were discussed in Section 5.) The *function class loader* is responsible for finding class files for implementations of external functions called within the EventScript program. (As explained in footnote 1 on page 6, these implementations are classes implementing the `FunctionBody` interface.) The function class loader is a pluggable component because the places to be searched for these class files depend on the surrounding environment. The *time provider* keeps track of the current date and time, handles timer start and timer stop actions,[19] and, upon the expiration of a timer, triggers a time event by constructing an

[19] A given computation instance (corresponding to a given grouping-key value) executes a strictly alternating sequence of timer start and timer stop actions. We can think of there being one timer for each computation instance, which is only started when it was previously stopped, and only stopped when it

event object for the time event and passing it to the same method that the input adapter invokes to feed events to the EventScript engine. The built-in function `now` obtains the current time and date from the time provider. Different implementations of the time provider might correspond to different time standards (e.g., UTC or the local time), or to simulated time.

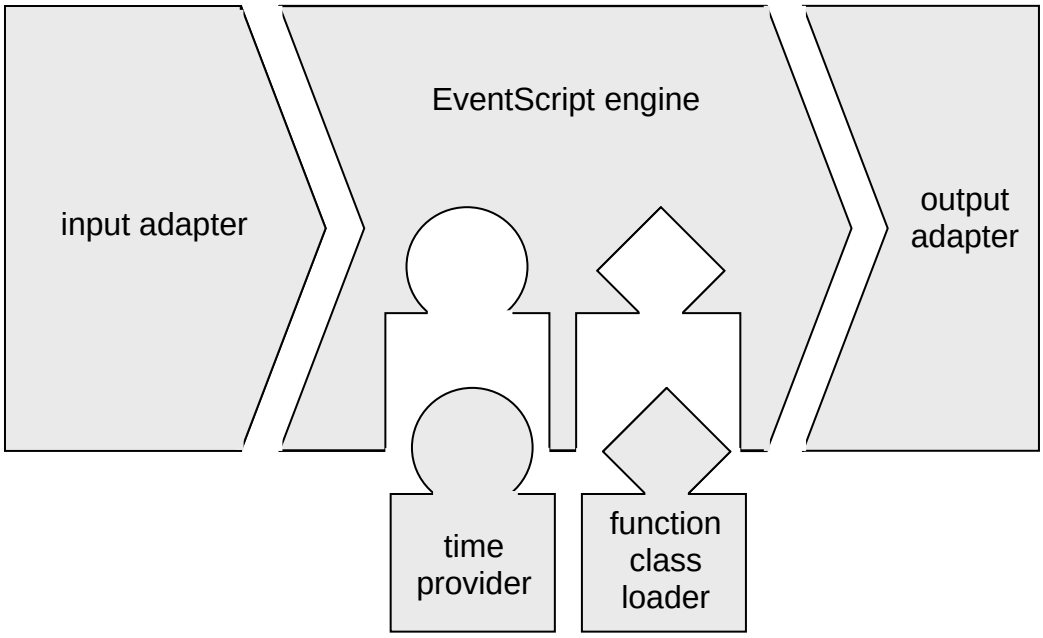

Fig. 7. Pluggable architecture of the EventScript run-time environment.

6.2 EventScript editor

The EventScript editor is a plug-in to the Eclipse workbench. It highlights EventScript keywords and finds matching pairs of parentheses, braces, and brackets. The editor interfaces with the EventScript compiler to mark the source code instantly

---

was previously started. Although there is always at most one pending timer expiration for each computation instance, timer expirations for different computation instances may be pending at the same time. Our timer-provider architecture has two layers. The bottom layer handles a single pending timer expiration at a time. The top layer implements multiple timers by maintaining a priority queue of all pending timer expirations but the earliest and submitting the earliest pending timer expiration to the bottom layer. If a computation instance submits a timer start action to the top layer while the bottom-layer timer is active, the top layer compares the expiration time for the timer start action with the expiration time currently pending in the bottom layer. If the expiration time in the timer start action is earlier, the bottom-layer timer is stopped, the expiration time that had previously been pending in the bottom layer is added to the top-layer priority queue, and the bottom-layer timer is restarted with the time in the timer start action; otherwise, the expiration time in the timer start action is simply added to the top-layer priority queue. If a timer stop action is received corresponding to an expiration time in the top-layer priority queue, the expiration time is removed from the priority queue. If a timer stop action is received corresponding to the expiration time pending in the bottom layer, the bottom-layer timer is stopped; then, if the top-layer priority queue is not empty, the earliest expiration time in the priority queue is removed and the bottom-layer timer is restarted with that time. The top layer maintains a record of which computation instance is responsible for each pending expiration time, so that time events can be delivered to the appropriate computation instance.

with syntax errors, semantic errors, and warnings; these syntax errors, semantic errors, and warnings also show up in the Problems View of the Eclipse workbench.

6.3 EventScript compiler

The compiler parses EventScript source, reporting syntax errors or constructing an abstract syntax tree if there are no syntax errors. The parser recognizes C-like include directives, which allow text in a single file to be included in multiple programs. Include files are intended to facilitate the construction of libraries of data-type and function declarations.

If an abstract syntax tree is successfully constructed, the compiler then makes several passes over the tree, to perform the following functions:

- checking for and issuing warnings about ambiguous regular expressions
- replacing each identifier representing the current event in an event-case clause, the current event in a group clause, or the loop index of a repeated action with a different syntax-tree node signifying this role
- inferring types of variables, checking for type inconsistencies, and reporting type inconsistencies as errors
- inserting implicit conversions of `long` operands of binary operators to type `double` if the other operands of those operators are of type `double`
- identifying the explicit function declaration or the built-in function referenced by each function call
- checking for cyclic paths through the regular expression that contain actions but no event marker (see footnote 8 on page 19), and reporting such paths as errors
- checking for and warning about possible references to uninitialized variables and assignments to variables that are not referenced
- inserting timer start actions and timer stop actions at appropriate places in the abstract syntax tree for the regular expression
- numbering all actions (including actions inserted in the previous pass) in order of occurrence (for later use in determining the order in which actions simultaneously reachable along different paths should be executed)
- performing simplifications, for example, flattening sequence regular expressions containing nested sequence regular expressions, alternative regular expressions containing nested alternative regular expressions, and intersection regular expressions containing nested intersection regular expressions
- transforming the abstract syntax tree into a normal form in which, for example, each wildcard event marker is replaced by an alternative regular expression with alternatives consisting of event markers for each named input event
- checking for potentially nonterminating programs that have grouping clauses, and warning about a possible resource leak in such programs (see Section 3.2)

The resulting abstract syntax tree is then used to build a nondeterministic finite automaton (NFA), based on the approach introduced by Thompson [Tho68] and elaborated upon by Aho, Sethi, and Ullman [Aho86]. The NFA is a graph consisting of nodes connected by arrows. Each arrow is labeled with either an event symbol (denoting a transition triggered by the arrival of that event from the state at the tail of the arrow to the state at the head of the arrow) or *epsilon* (denoting an immediate transi-

tion without the arrival of another event symbol), as well as with a sequence of zero or more actions to be executed as part of the transition. For each kind of regular expression operator, there is a method that recursively constructs a corresponding subgraph of the NFA containing one entry node and one exit node, as shown in Fig. 8. To generate the NFA for an intersection regular expression such as `R & S & T`, we take advantage of DeMorgan's Law for regular expressions: `R & S & T` = `~(~R | ~S | ~T)`, where, if *E* is a regular expression, *~E* means `.* - E`. To construct the subgraph for `~R`, we convert the subgraph for `R` into a *deterministic* finite automaton (in a manner that we will describe in the next paragraph), then change each accepting state to a rejecting state and *vice versa* to obtain a deterministic finite automaton for `~R`, then add an epsilon transition from each accepting state to a new single exit node to obtain a single-entry/single-exit nondeterministic subgraph for `~R`. We do the same for `S` and `T`, then combine the subgraphs for `~R`, `~S`, and `~T` using the usual construction for alternative regular expressions (Fig. 8(e)). Finally, we obtain the subgraph for `~(~R | ~S | ~T)` from the subgraph for `~R | ~S | ~T` in the same way that we obtained the subgraph for `~R` from the subgraph for `R`. We generate the NFA for a difference regular expression such as `R-S` in a similar manner, taking advantage of the fact that `R-S = R & ~S = ~(~R | S)`.

For example, let us revisit the keypad example introduced in Section 3.3.1:

```
in long Key case { Key==0 ? K0 : Key==7 ? K7 : Other }
out {} Unlock

( .* K0() K0() K7() { !>Unlock({}); } )*
```

Fig. 9 shows the NFA generated for this program.

Next, the compiler transforms the NFA into another NFA that has no epsilon transitions, but has zero or more initial actions and one or more start states. We classify each state of the NFA as *significant* if it has at least one non-epsilon arrow leaving it, or no arrows leaving it at all, and *insignificant* otherwise. Fig. 10(a) shows the significant and insignificant states of Fig. 9. Our transformation eliminates all insignificant states. An *acyclic epsilon path* from a node n_0 to a node n_k, $k \geq 0$, is a sequence of distinct nodes n_0, ..., n_k such that, for $0 < i \leq k$, there is an epsilon arrow from n_{i-1} to n_i. For each acyclic epsilon path from the starting state to a significant state *s*, we declare *s* to be a starting state of the transformed NFA, and we include the sequence of actions encountered along that acyclic epsilon path in the initial actions of the transformed NFA. For each non-epsilon arrow from a significant state s_1 to some other (significant or insignificant) state s_2, we find all acyclic paths consisting only of zero or more epsilon arrows from s_2 to some significant state s_3. (If s_2 is significant, these paths include the path of length zero whose only node is s_2.) In place of the arrow from s_1 to s_2 we insert, for each such path, an arrow from s_1 to s_3, labeled with the same event symbol as the deleted arrow from s_1 to s_2, and with the sequence of actions encountered along the path from s_2 to s_3. Fig. 10(b) shows the epsilon-free NFA for the keypad program.

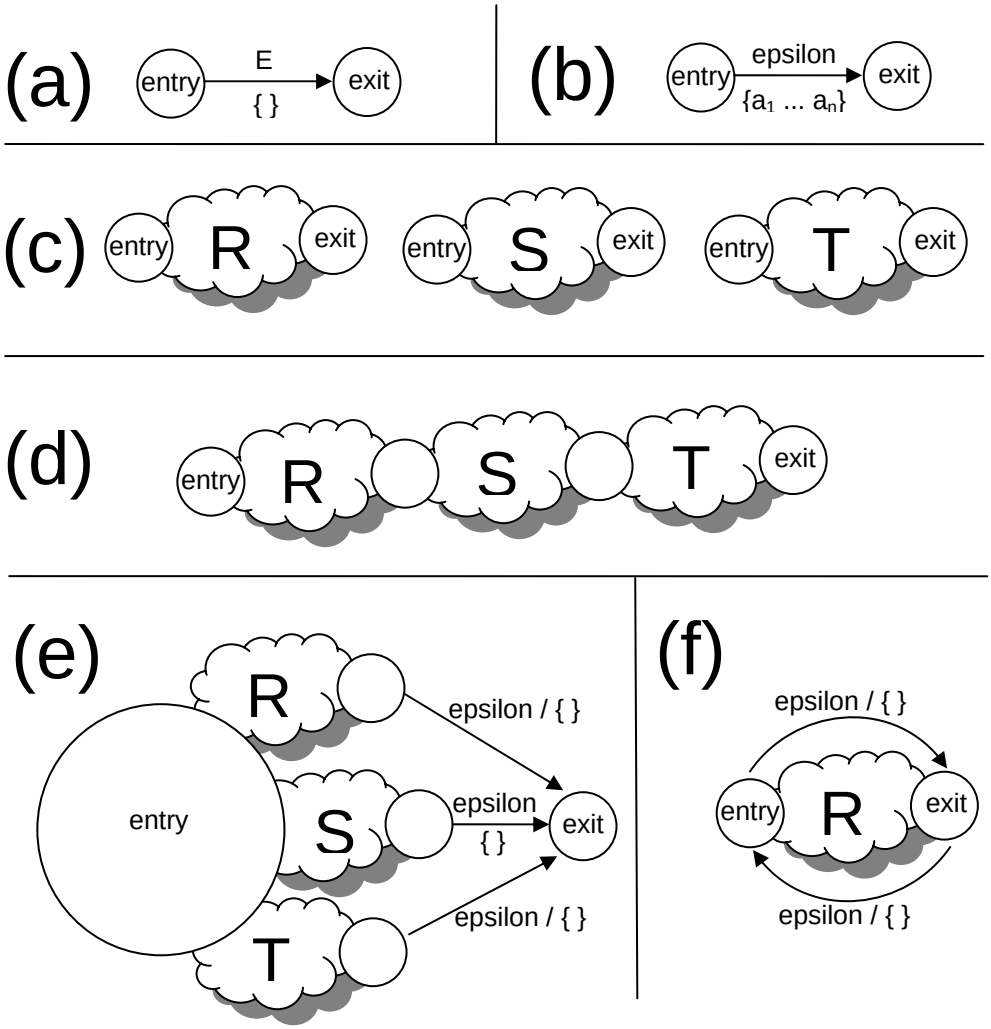

Fig. 8. Construction of a nondeterministic finite automaton from a regular expression. The subgraph for an event marker for event E consists of and entry node and an exit node connected by a single arrow labeled E, with an empty sequence of actions, as shown in (a). The subgraph for an event block **{** a_1, ..., a_n **}** consists of and entry node and an exit node connected by a single arrow labeled *epsilon*, with athe sequence of actions a_1, ..., a_n, as shown in (b). In (c), suppose the cloud labeled R, together with the explicitly labeled entry and exit nodes, form the subgraph for a regular expression *R*, and similarly for S and T. Then (d) shows the subgraph for the sequence regular expression *R S T*, (e) shows the subgraph for the alternative regular expression *R* **|** *S* **|** *T*, and (f) shows the subgraph for the repetition regular expression R^*. (In (d), the same node generated as the exit node for the *R* subgraph serves as the entry node for the *S* subgraph, and the same node generated as the exit node for the *S* subgraph serves as the entry node for the *T* subgraph. In (e), a single node serves as the entry node for the *R*, *S*, and *T* subgraphs.)

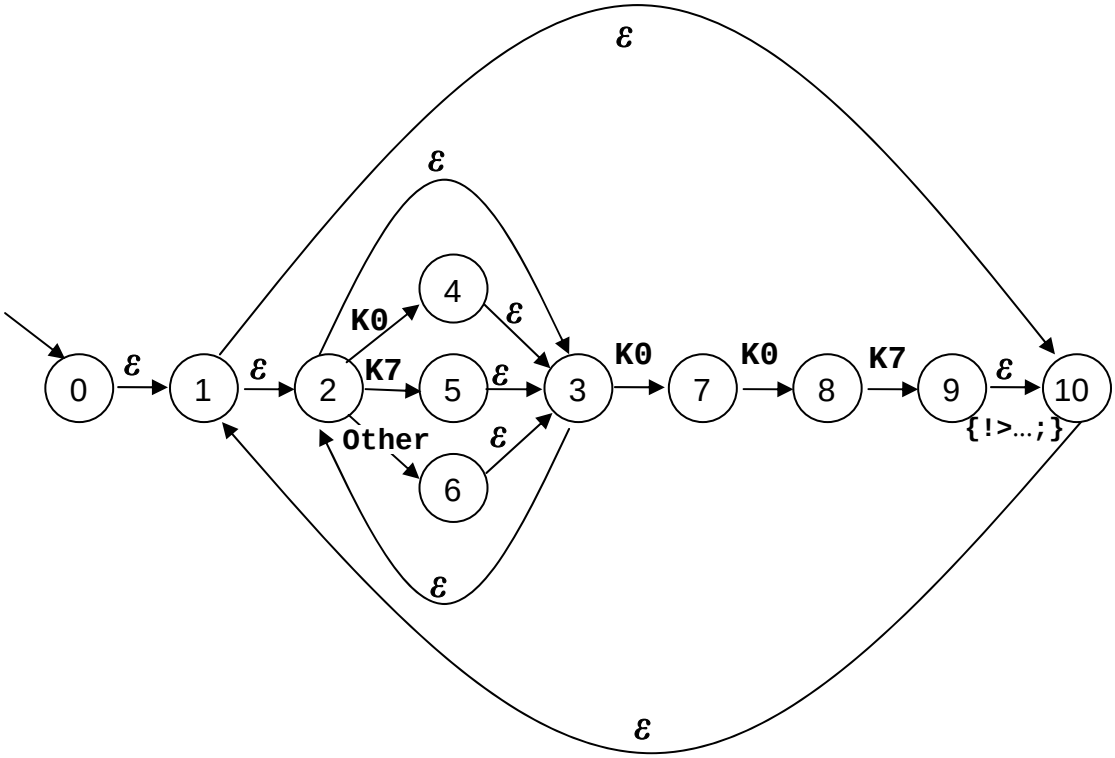

Fig. 9. NFA generated for the keypad example. Epsilon transitions are marked with the symbol ε. Abstract-syntax-tree transformations have replaced the regular expression `.` with **`(K0|K7|Other)`** before the generation of the NFA. The subgraph generated from this alternative regular expression consists of states 2 through 6, the transitions from state 2 to states 4, 5, and 6, and the transitions from states 4, 5, and 6 to state 3. The subgraph generated from the repetition **`.*`** (i.e., **`(K0|K7|Other)*`**) consists of this subgraph plus the epsilon transitions from state 2 to state 3 and from state 3 to state 2. The subgraph generated from the sequence **`.* K0() K0() K7() { !>Unlock({}); }`** consists of this subgraph plus states 7 through 10 and the transitions from state 3 to state 7, state 7 to state 8, state 8 to state 9, and state 9 to state 10. The subgraph for the entire outer repetition consists of this subgraph plus the epsilon transitions from state 1 to state 10 and from state 10 to state 1. The entire NFA consists of the starting state 0 plus an epsilon transition from there to the subgraph for the regular expression.

Following Rabin [Rab59], we then convert the epsilon-free NFA into a deterministic finite automaton (DFA), each of whose states corresponds to a set of states of the NFA.[20] The initial actions of the DFA consist of the initial actions of the transformed NFA. The initial state of the DFA corresponds to the set of start states of the transformed NFA. To compute the transition on input *x* from a DFA state corresponding to a set N_d of NFA states, we follow all the NFA arrows labeled *x* that emerge from states in N_d. The target DFA state corresponds to the set of NFA states reached by these NFA arrows. The actions associated with the DFA transition are the actions associated with these NFA arrows, ordered according to their original positions in the source program. Fig. 11 shows the DFA for the keypad program.

The compiler writes the XML file, described in Section 6.1, that is read by the EventScript engine. The transition table in this file contains the transitions of the DFA constructed from the epsilon-free NFA.

[20] In the absence of actions, Rabin's method can be applied to an NFA with epsilon transitions. Our purpose in eliminating epsilon transitions is to ensure that actions are handled properly.

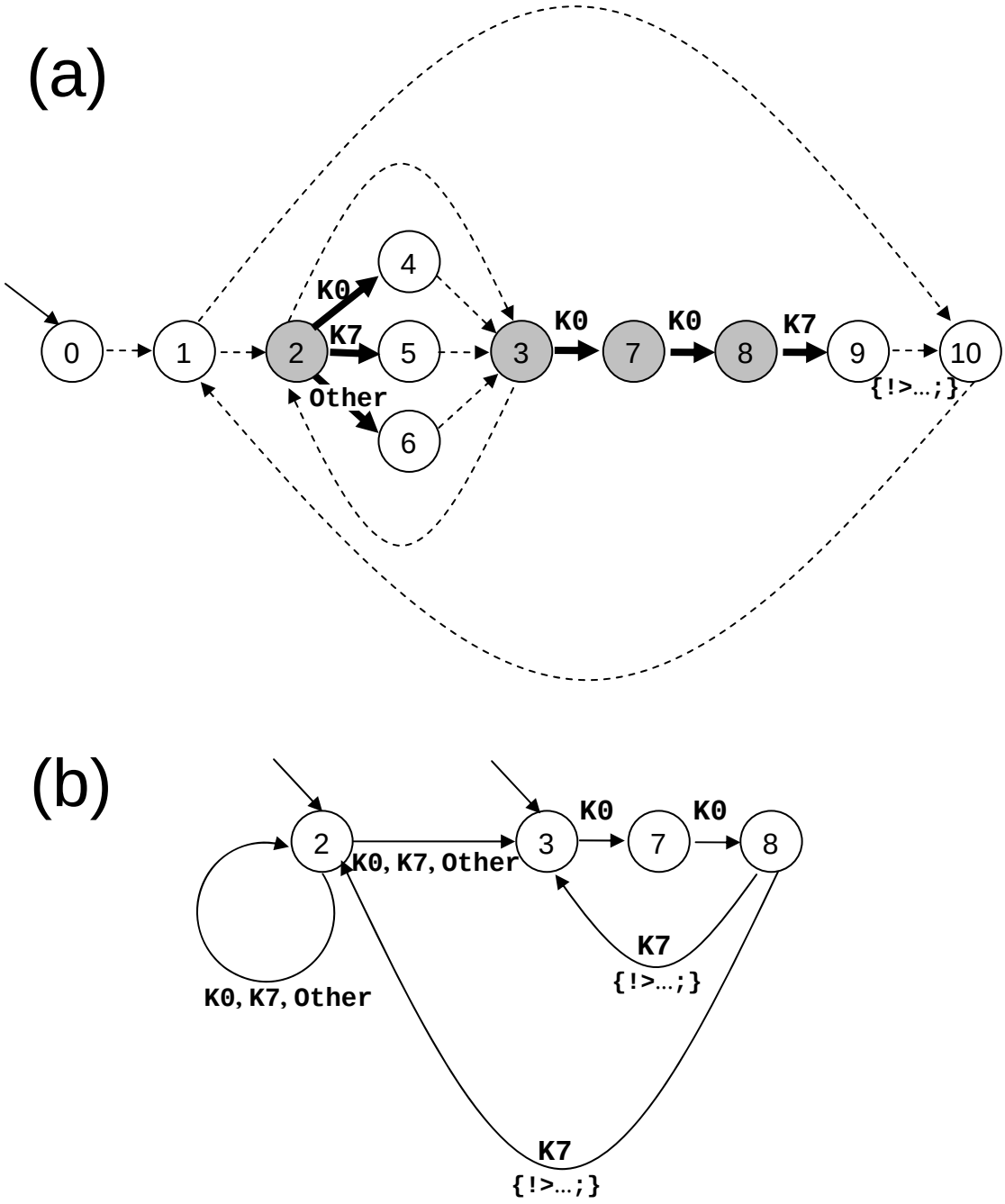

Fig. 10. Removal of epsilon transitions from the NFA of Fig. 9. In (a), we have marked the epsilon transitions with unlabeled dotted lines and shaded in the significant states (i.e., those with non-epsilon transitions coming out of them. In (b), we have eliminated epsilon transitions and insignificant states. Because of the acyclic epsilon path from state 4 to state 3, we have replaced the transition on input **`K0`** from state 2 to state 4 with a transition from state 2 directly to state 3. The transitions from state 2 on inputs **`K7`** and **`Other`** are handled similarly. Because of the acyclic epsilon paths from state 9 to state 10 to state 1 to state 2 and from state 9 to state 10 to state 1 to state 2 to state 3, we have replaced the transition from state 8 to state 9 on input **`K7`** with transitions from state 8 directly to states 2 and 3. The emit action encountered along both these paths is associated with the new transitions. Because of the acyclic epsilon paths from state 0 to state 1 to state 2 and from state 0 to state 1 to state 2 to state 3, states 2 and 3 are both starting states of the transformed NFA.

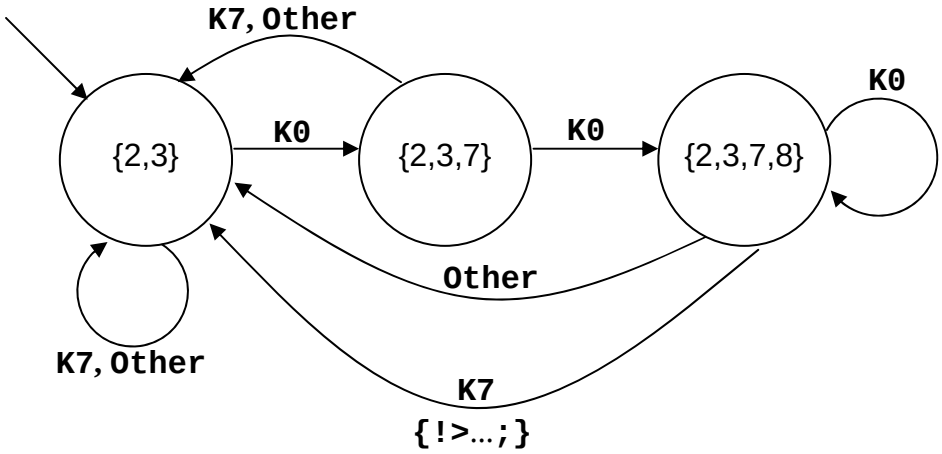

Fig. 11. The DFA for the keypad program. Each DFA state is labeled with the set of NFA states to which it corresponds.

6.4 Hierarchical machines

The number of states in the NFA generated for a given regular expression is roughly proportional to the size of the regular expression. However, since each state of the DFA constructed from an NFA corresponds to a distinct set of NFA states, it is theoretically possible for an NFA with n states to have a corresponding DFA with 2^n states. A well-known example of this exponential blowup is reflected in the following EventScript program, in which *k* is a placeholder for some integer constant:

```
in {} A, {} B
out { } Z

.* A() .{k} { !>Z({}); }
```

This regular expression matches a string of any zero or more input events, followed by an `A` event, followed by any k input events. Thus the emit action is reachable whenever the $(k+1)^{th}$ most recent input event is an `A` event. It follows that the DFA must, in effect, remember the history of the $k+1$ most recent input events. Since 2^{k+1} distinct histories are possible, the DFA must have at least 2^{k+1} states.

We have yet to encounter a real-world problem for which the DFA size is problematic. Nonetheless, we felt it was important to have a fallback mechanism for any such cases that may arise. That fallback consists of a distinct variety of state machine that we call a *hierarchical machine*. The size of a hierarchical machine is proportional to the size of the regular expression from which it was generated. However, the time for a hierarchical machine to process an input event (not counting the time spent executing actions) is also proportional to the size of the regular expression; for a DFA, this time is independent of the regular expression (and quite small).

The EventScript compiler has an alternative mode in which, rather than generating a DFA, it generates a hierarchical machine, and writes an XML file that contains a specification of this hierarchical machine rather than a specification of a DFA. The passes of the compiler that manipulate the abstract syntax tree (which contain the bulk of the semantic analysis performed by the compiler) are the same regardless of the compiler mode; the only difference between the modes is the manner in which the fi-

nal version of the abstract syntax tree is used to generate an XML file. The outermost element of the XML file written by the compiler has an attribute indicating whether it contains a specification of a DFA or a specification of a hierarchical machine. When the EventScript engine loads the XML file, it checks this attribute. If the file contains the specification of a hierarchical machine, the EventScript engine constructs and executes that hierarchical machine. The interface for feeding events to the EventScript engine is the same regardless of whether the EventScript engine is executing a DFA or a hierarchical machine. The handling of event classification and event grouping is the same in either case, although the form of instance-context objects is different.

A hierarchical machine is an object that may contain other hierarchical machines within it, and whose structure mirrors the structure of the regular expression that it matches. There are different subclasses of hierarchical machines for different kinds of regular-expression operators. Fig. 12 illustrates the structure of the hierarchical machine for a particular regular expression. Each hierarchical machine is responsible for tracking the possible current positions of the matching process within its part of the regular expression. A hierarchical-machine object has a method, implemented differently for each hierarchical-machine subclass, that processes the arrival of an event and returns a boolean value indicating whether the event can be interpreted as completing a match of its part of the regular expression. A hierarchical machine that contains inner hierarchical machines works by passing the just-arrived event to each inner machine, along with a flag indicating whether the matching process is potentially positioned at the beginning of the inner machine's part of the regular expression. Whenever a sequence machine determines that the matching process is potentially positioned at an action, it executes that action.

Unfortunately, as far as we are aware, there are no procedures or heuristics for examining a regular expression and determining quickly whether its DFA will have an intractably large number of states. Therefore, the EventScript compiler cannot automatically choose to compile a particular program into a hierarchical machine instead of a DFA. Rather, the compiler tries by default to compile a DFA, but terminates the compilation and issues an error message when a predetermined limit on the number of states is exceeded.[21] The EventScript developer can then reinvoke the compiler with an option specifying a higher limit or with an option specifying that a hierarchical machine should be produced.

We have considered the possibility of a *hybrid hierarchical machine*, in which some of the inner hierarchical machines, responsible for particular subexpressions of the regular expression, are implemented using DFA transition tables for those subexpressions. These inner machines, rather than tracking a single current DFA state, track a set of possible current DFA states. When one of these inner machines is invoked with a flag indicating that the matching process is potentially positioned at the beginning of its part of the regular expression, the starting state of its DFA is added to the set of possible current states. One heuristic strategy for the construction of a hybrid machine is to attempt to compile a DFA for the entire regular expression and, whenever the DFA state limit is reached, abandoning that attempt and attempting instead to compile a hierarchical machine in which the same strategy is applied recursively to attempt to compile each immediate subexpression of the regular expression as a DFA.

[21] The default limit is set at 40,000. On our 2 GHz Pentium M with 2GB of RAM, the compiler runs for about 9.4 seconds before reaching this limit and terminating the compilation.

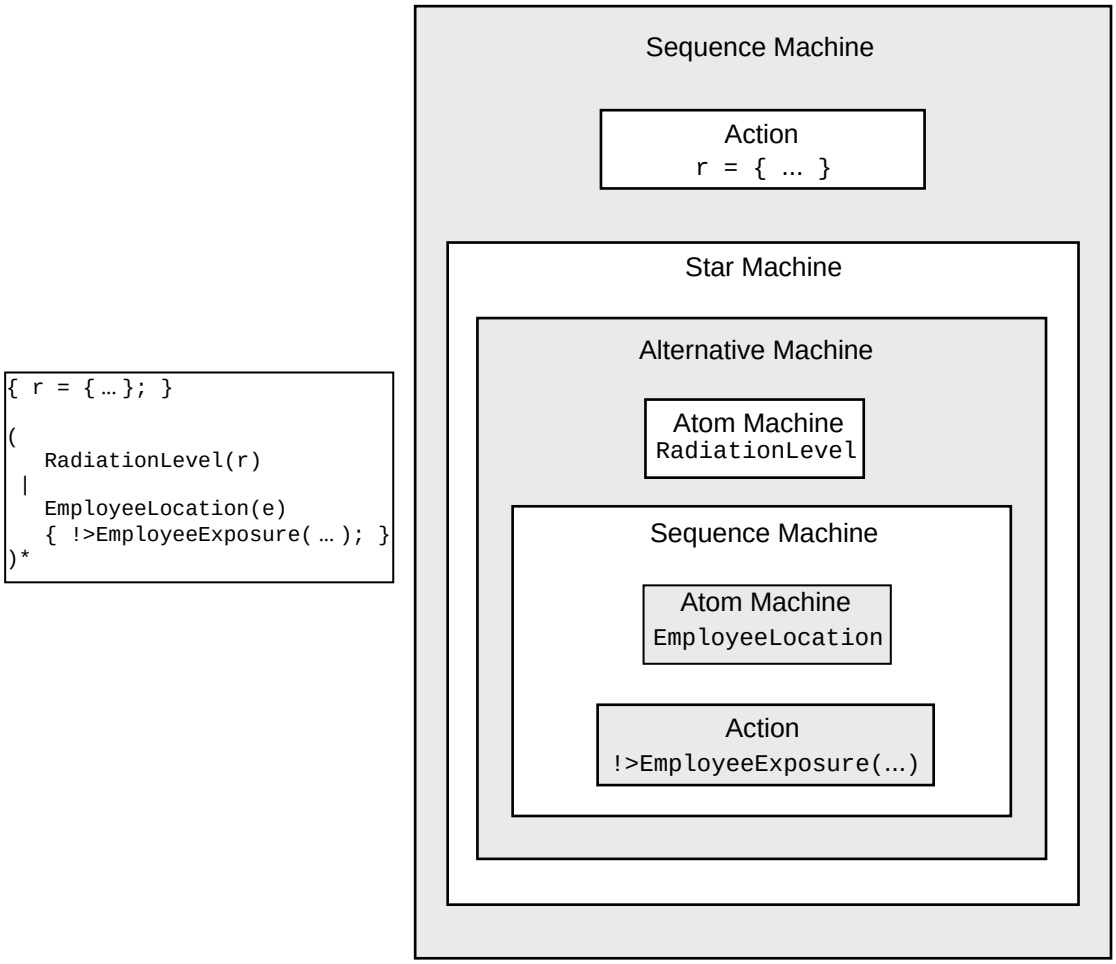

Fig. 12. A regular expression and its hierarchical machine.

6.5 EventScript tester

Events arrive from the outside world at arbitrary times, so anomalous behavior of an event-processing application is typically not reproducible. The EventScript tester facilitates testing and debugging by enabling EventScript programs to be executed in a reproducible manner against a *timed event trace*. The execution runs in simulated time. The EventScript tester reports emitted events on an output console, indicating both the time at which the event was emitted and the value carried by the event.

A timed event trace is a sequence of *event-occurrence items*, each specifying that an event with a given name and value should arrive; and monotonically increasing *timestamps*, each specifying that the simulated time should be advanced to a particular date and time. Here is a sample fragment of a timed event trace:

```
[1979-03-28 4:00:00:000] RadiationLevel("Rm400",0.013)
RadiationLevel("Rm405",0.021) RadiationLevel("Rm410",0.018)
RadiationLevel("Rm415",0.020) RadiationLevel("Rm420",0.024)
[4:00:07:500] RadiationLevel("Rm400",0.012)
RadiationLevel("Rm405",0.023) RadiationLevel("Rm410",0.020)
RadiationLevel("Rm415",0.023) RadiationLevel("Rm420",0.026)
[4:00:15:000] RadiationLevel("Rm400",0.013)
RadiationLevel("Rm405",0.022) RadiationLevel("Rm410",0.020)
RadiationLevel("Rm415",0.026) RadiationLevel("Rm420",0.030)
[4:00:22:500] RadiationLevel("Rm400",0.014)
RadiationLevel("Rm405",0.027) RadiationLevel("Rm410",0.023)
RadiationLevel("Rm415",0.028) RadiationLevel("Rm420",0.038)
```

Timed event traces might be created manually by people testing or debugging programs, created automatically by test-case-generation programs, or captured by monitoring actual executions or scraping execution logs.

To validate the event-occurrence items in the timed event trace, and to generate event objects from them, the EventScript tester needs the information in the input event declarations of the program being tested. The EventScript compiler has an invocation option directing it to generate, in addition to the XML file describing the finite-state machine, an XML file containing metadata about the declared input events. The EventScript tester can be invoked with a specified state-machine XML file, a specified metadata XML file, and a specified timed-event-trace file. Alternatively, it can be invoked with a specified EventScript source file and a specified timed-event-trace file, in which case the tester invokes the EventScript compiler to obtain the state-machine and metadata XML files.

The internal architecture of the EventScript tester, shown in Fig. 13, exploits the pluggable architecture of the EventScript run-time environment, shown in Fig. 7. A single component, the timed-event-trace processor, reads the timed trace and plays the roles of both the input adapter and the time provider in Fig. 7. Specifically, the timed-event-trace processor processes an event-occurrence item by constructing an EventScript event object and by calling a method that takes an event object as a parameter to feed the event object to the EventScript engine; it processes a timestamp by advancing the simulated time and delivering time events for any pending timers that are set with an expiration time less than or equal to the new simulated time.[22] Calls by the EventScript program on the built-in function `now` are passed through the standard time-provider interface, and return the simulated time. The role of the output adapter in Fig. 7 is played by a component that processes each event fed to it by printing the simulated time, the event name, and the event value on the EventScript tester output console.

[22] The processing of a time event may itself cause the EventScript program to execute new timer start or timer stop actions. Therefore, upon processing a timestamp, the timed-event-trace processor repeatedly looks for the *earliest* pending timer-expiration time less than or equal to the time in the timestamp, advances the simulated time to that expiration time, and feeds the resulting time event to the EventScript engine; the repetition ends when there is no longer a pending timer-expiration time less than or equal to the time in the timestamp. At this point the simulated time is advanced to the time in the timestamp, completing the processing of the timestamp.

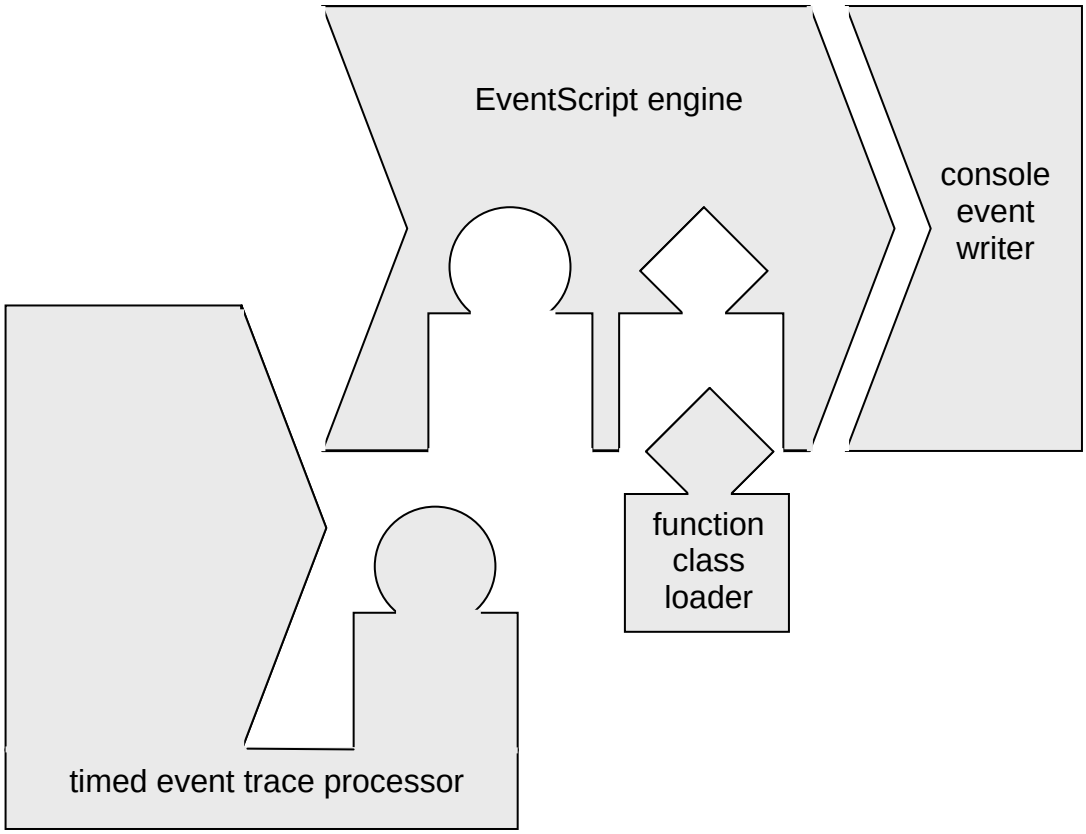

Fig. 13. Architecture of the EventScript tester.

The architecture of the tester would make it easy to construct other testing tools out of the same building blocks, although we have not yet done this. For example, the role of the output adapter could be filled by a component that compares emitted events to a file of expected test outputs, to facilitate automated testing. Alternatively, a pipeline of EventScript engines could be interposed between the timed event trace processor and the console event writer, allowing all stages of a pipelined EventScript solution like those presented in Section 4.5 to be tested together.

6.6 EventScript in DRIVE

EventScript has been incorporated in the Distributed Responsive Infrastructure Virtualization Environment, or DRIVE [Che07]. DRIVE is an environment for developing, testing, and deploying event-based applications. A DRIVE solution is built out of *components* that may consume events through named input ports and may emit events through named output ports. A component may be implemented as a *composite* component, in which case events arriving through input ports of the composite component are fed to input ports of some subcomponents, events emitted from the output ports of some subcomponents are fed to the input ports of other subcomponents, and events emitted from the output ports of some subcomponents are emitted through output ports of the composite component. Alternatively, a component may be implemented as an *atomic* component. The logic of an atomic component may be specified either in Java or in EventScript.

For an atomic component implemented in EventScript, EventScript input event names correspond to DRIVE input port names and EventScript output event names correspond to DRIVE output port names. A DRIVE event consists of a value, analogous to the value carried by an EventScript event.[23] When a DRIVE event arrives at a given input port of an EventScript atomic component, that component's EventScript engine receives an event whose name is the name of the input port and whose value is the DRIVE event value. When the DRIVE component's EventScript engine emits an output event, a DRIVE event with the value carried by the EventScript output event is emitted from the DRIVE port whose name is the name of the EventScript output event.

Some DRIVE atomic components correspond to event producers and consumers in the real world, such as sensors, actuators, graphical user interfaces, and web services. DRIVE provides a valuable bridge between these real-world entities and EventScript programs. The DRIVE run-time environment includes one standard input adapter and one standard output adapter, applicable to all DRIVE atomic components implemented in EventScript. Thus the user of EventScript in DRIVE need not write any EventScript adapters. DRIVE domain-specific component libraries include predefined atomic components for certain real-world producers and consumers, such as RFID readers and signal lights. Application developers can write their own Java atomic components for other producers and consumers.

6.7 Java API for functions and adapters

We have defined and implemented a Java API that provides the facilities needed to write external functions in Java and to incorporate the EventScript engine in a new run-time environment.

The facilities needed to write external functions in Java include the declaration of the `FunctionBody` interface (see footnote 1 on page 6), methods to obtain Java values from `EventScriptValue` objects representing values in the EventScript type system, and methods to construct `EventScriptValue` objects from Java values. Here is an example of these facilities at work to implement a function that retrieves the value of a system environment variable with a given name:

```
package myFuncs;
import com.ibm.es.external.EventScriptValue;
import com.ibm.es.external.FunctionBody;
public final class EnvVar implements FunctionBody {
  public EventScriptValue evaluate(EventScriptValue[] args) {
     String envVarName = args[0].getStringValue();
     String envVarValue = System.getenv(envVarName);
     return EventScriptValue.newStringValue(envVarValue);
  }
}
```

[23] Work is currently in progress to convert DRIVE values to be based on EventScript data types. For now, a DRIVE value is simply a Java object. The DRIVE input adapter converts Java objects to corresponding EventScript values and the DRIVE output adapter converts EventScript values to corresponding Java objects.

The `evaluate` method converts the function argument from the `EventScript-Value` representation of an EventScript `string` value to the Java class `String`, passes that `String` object to the Java method `System.getenv` to obtain a Java `String` result, converts that result to the `EventScriptValue` representation of an EventScript `string` value, and returns that `EventScriptValue` object.

Part of the task of incorporating the EventScript engine in a new run-time environment is the writing of adapters. Facilities needed to write adapters include the methods we have already seen for converting between `EventScriptValue` objects and Java values, as well as a constructor for creating an `EventScriptEvent` object with a given name and `EventScriptValue`, and methods for extracting the name and `EventScriptValue` from an `EventScriptEvent`. Other facilities needed to incorporate the EventScript engine in a new run-time environment include a method to read a state-machine XML file of the form generated by the EventScript compiler and generate a corresponding `StateMachine` object (which may be the representation of either a DFA or a hierarchical machine); and a constructor to create an `EventScriptEngine` object with a specified `StateMachine`, a specified `ClassLoader` for external-function classes, and a specified consumer for emitted events. The consumer is an object implementing the interface `EventConsumer`, which has the following method:

```
void consume(EventScriptEvent event)
```

Both output adapters and the `EventScriptEngine` class implement this interface, so the `EventConsumer` supplied to the `EventScriptEngine` constructor might be an output adapter or another `EventScriptEngine`. Once an `EventScriptEngine` object is constructed, the task of the input adapter is simply to construct an `EventScriptEvent` object and pass it to the `consume` method of the `EventScriptEngine` whenever it determines that there is a new input event.

7. Future directions

As EventScript gains wider use, by people with a wider variety of needs and outlooks, we will gain new insights. We will learn which tasks EventScript excels at, and which tasks are best addressed by other approaches. We will learn which concepts are natural to EventScript programmers and which are confusing, and we will learn better ways to explain the language in or teaching materials and error messages. If we learn of language idiosyncrasies that make the programmer's task more difficult, we will study ways to remedy those problems in a later version of the language. We expect EventScript programmers to surprise us with clever new programming techniques, beyond those we discussed in Section 4.

It may be that for certain programming problems, or for visually-oriented programmers, it will be easier to specify the regular-language model of an input event stream by constructing a (possibly nondeterministic) state-transition diagram with a graphical editor than by writing the text of a particular regular expression. Actions, grouping-key expressions, and event-classification predicates would still be specified textually. Some state diagrams can only be transformed into regular expressions by writing certain subexpressions multiple times, and Bruggemann-Klein and Wood

[Bru98] proved that there are deterministic state diagrams for which no unambiguous regular expression can be written. These facts suggest that state diagrams can, in some sense, be a more powerful form of expression than regular expressions. For example, the time-debouncing program on page 29 is probably more easily specified and understood in terms of the diagram in Fig. 14 than in terms of a regular expression. Much of the EventScript compiler, and all of the EventScript run-time environment, would be applicable to programs entered in the form of state-transition diagrams.

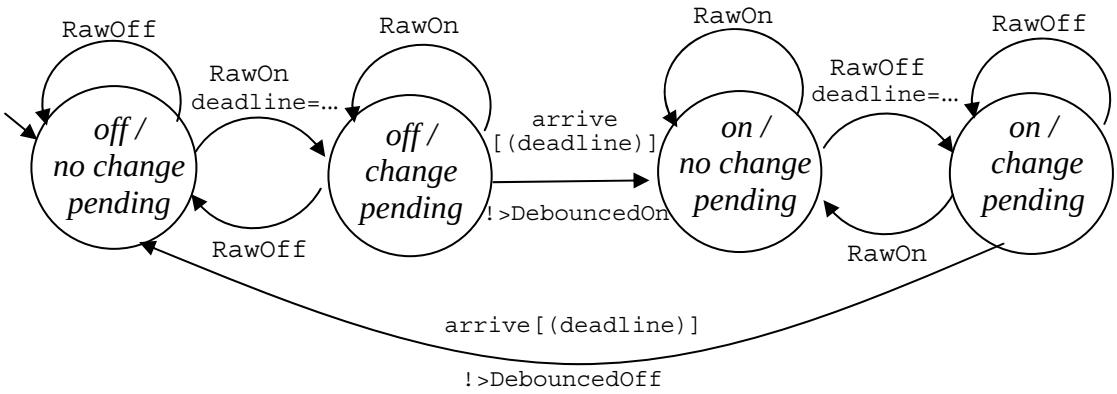

Fig. 14. Deterministic state diagram for the time debouncing program.

The rapid and wide adoption of Java can be attributed in part to the large number of useful classes in the standard Java library. Over time, we expect to build libraries of data types and external functions that will make EventScript a more powerful and convenient programming tool for particular domains. We have already written a package of computational geometry data types and external functions that are useful for recognizing spatio-temporal patterns (for example, for receiving location-report events in terms of an *x*-*y* coordinate system and reporting that people or objects have entered or left certain zones defined as circles or polygons). As the DRIVE environment described in Section 6.6 transitions to the EventScript type system, it will be useful to devise a repository of data-type definitions that can be used throughout the DRIVE environment and also incorporated in EventScript programs.

There are many ways in which EventScript development tools could be better integrated with the Eclipse workbench. The existing EventScript editor could be enhanced to provide syntax completion. The EventScript compiler could be invoked automatically when an EventScript source file is saved. The EventScript tester could be made directly invocable from Eclipse run configurations.

Several tool improvements independent of Eclipse are also possible. The EventScript compiler could be enhanced to compile to hybrid machines, as discussed in Section 6.4. (The required run-time support for hybrid machines has already been implemented.) Section 6.5 described two ways to enhance the EventScript tester: First, it could be enhanced to support the integrated testing of pipelines of EventScript programs, like those illustrated in Section 4.5. Second, it could be enhanced to automatically validate the emitted event sequence against a file containing expected outputs. Automatic validation, in turn, would facilitate the development of a JUnit-like tool to support the automated testing of large suites of EventScript programs.

EventScript will be more wide useful, and the industry will benefit, if EventScript programs can also be executed on platforms other than a Java Virtual Machine. There are many rule-based event-processing engines in the marketplace, and

the last phase of the EventScript compiler (which transforms the final version of the abstract syntax tree to an XML file describing a DFA or a hierarchical machine) could be replaced with a generator of rules for one of these engines. Moreover, translators and run-time modules could be devised to allow the transition tables currently emitted by the compiler to be interpreted by a C program. This would allow EventScript programs to be executed in embedded environments that do not support Java. The feasibility of this approach was proven by a summer intern at the IBM Thomas J. Watson Research Center, who write a C program to interpret the transition tables generated by the compiler for an earlier version of EventScript (without time events or event grouping), enabling EventScript programs to be executed on a Nokia 770.

Acknowledgements

Karl Trygve Kalleberg, then a doctoral candidate at the University of Bergen, Norway, made invaluable contributions to the design and implementation of EventScript while he was a summer intern at the IBM Thomas J. Watson Research Center in the summer of 2006. Karl Trygve wrote the initial version of the EventScript compiler, run-time system, and Eclipse plug-in, and brought valuable insights from the implementation effort to discussions on the design of the then still-evolving language. Among other valuable contributions, Karl championed the notion of hierarchical and hybrid machines that was presented in Section 6.4. Han Chen, the principal designer and implementer of DRIVE, did the work to integrate EventScript into DRIVE. David A. Wood, an early user of EventScript, helped clarify the ideas in Section 4.1 about a regular expression as a model of an input event stream, and the importance of being able to directly reflect specifications expressed in terms of patterns of events that should trigger particular actions. David was also the source of the proposal at the end of Section 6.5 for a version of the EventScript tester that would allow all stages of a pipelined solution to be tested in tandem.

References

[Rit74] Ritchie, Dennis M., and Thompson, Ken, 1974. The UNIX time-sharing system. *Commun. ACM* **17**, 7 (Jul. 1974), 365–375.

[Luc02] Luckham, David, 2002. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, Boston, Massachusetts, 2002.

[DeR75] DeRemer, Frank, and Kron, Hans, 1975. Programming-in-the-large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California, April 21 - 23, 1975). ACM Press, New York, 114–121.

[Fri06] Friedl, Jeffrey E.F., 2006. *Mastering Regular Expressions*, 3rd ed. Reilly, Sebastopol, California, 2006.

[Bru98] Bruggemann-Klein, Anne, and Wood, Derick, 1998. One-unambiguous regular languages. *Information and Computation* **142**, 2 (1998), 182–206.

[Rab59] Rabin, M.O., and Scott, D. Finite automata and their decision problems. *IBM Journal of Research and Development* **3**, 2 (April 1959), 114–125.

[Har87] Harel, David. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* **8** (1987) 231–274

[Jac75] Jackson, M.A. *Principles of Program Design*. Academic Press, London, 1975

[Tho68] Thompson, Ken, 1968. Regular expression search algorithm. *Commun. ACM* **11**, 6 (June 1968), 419–422.

[Aho86] Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986

[Che07] Chen, Han, Chou, Paul B., Cohen, Norman H., and Duri, Sastry. Extending SOA/MDD to sensors and actuators for sense-and-respond business processes. IEEE International Conference on e-Business Engineering (ICEBE 2007), Hong Kong, China, October 24-26, 2007, to appear