RC24517 (W0803-096) March 21, 2008
Computer Science

IBM Research Report

Compound Event Processing Using Regular Expressions: Examples from EventScript

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

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

Compound Event Processing Using Regular Expressions: Examples from EventScript

Norman H. Cohen
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
+1.914.784.7536
ncohen@us.ibm.com

ABSTRACT

Many formalisms have been proposed for specifying patterns of events and responses to the recognition of such patterns. Too often, these formalisms are intricate, unintuitive, and unfamiliar to typical programmers. We propose an alternative that is simple, intuitive, and familiar: regular expressions with placeholders for particular types of events, interleaved with actions that perform computations and emit output events. We have designed and implemented a language called EventScript that uses such regular expressions as the basis for building event-processing applications. Through dozens of short examples, spanning the spectrum from device-level events to cyber-physical system events to business events, we show that regular expressions with actions are a powerful and versatile basis for specifying event-processing logic.

Categories and Subject Descriptors

C.3 [**Special Purpose and Application-Based Systems**]: *Real-time and embedded systems;* D.2.2 [**Software Engineering**]: Design Tools and Techniques – *State diagrams;* D.3.3 [**Programming Languages**]: Language Constructs and Features – *Patterns;* H.2.4 [**Database Management**]: Systems – *Rule-based databases;* I.5.1 [**Pattern Recognition**]: Models – *Structural;* J.1 [**Administrative Data Processing**]: *Business*; J.7 [**Computers in Other Systems**]: *Industrial control; Process control; Real time*

General Terms

Algorithms, Design, Languages

Keywords

event processing; reactive programs; event patterns; regular expressions

1. INTRODUCTION

Many previous approaches for specifying patterns of events are extremely intricate, providing a dizzying range of unfamiliar operators, modes, and options. We assert that a much simpler and more familiar approach—the use of regular expressions to specify patterns of events—is sufficient for a wide variety of event-processing applications.

Event patterns arise at many levels of event-processing systems. Close to the hardware, event patterns can be used to interpret raw sensor input as an indication of a physical occurrence or condition—for example, to sample sensor signals periodically or to report a set of closely spaced signals from an RFID tag as a single tag reading. In a cyber-physical system, event patterns can be applied to indications of the physical occurrences to detect situations meaningful to an application, such as a combination of signals from light-beam detectors and RFID readers indicating that a package on a conveyor belt is missing an RFID tag, or computing the average time it takes for car-mounted RFID tags to get from one point on a highway to the next. At the business-process level, event patterns can be applied to sequences of business events produced by business IT systems, to maintain inventories at appropriate levels or monitor suspicious patterns of ATM withdrawals. Event-processing systems at these various levels have different nonfunctional requirements: Near the hardware level, real-time response is important, and near the business-process level, recoverability without data loss is important. However, we assert that the same kinds of compound-event patterns are useful at each level. The single regular-expression model we propose here can be implemented by various systems with various nonfunctional characteristics.

We have defined and implemented a language called EventScript for writing regular expressions that match sequences of incoming events. Placeholders in these regular expressions match incoming events of particular event types. Actions performing computations and emitting output events can be embedded within the regular expression. EventScript also provides for events triggered by the passage of time and the grouping of events into separate event streams, each of which is matched independently against the same regular expression, based on the contents of the incoming events.

A previous paper [5] focuses on details of the EventScript language and on programming-language design and implementation issues. Our focus here is on paradigms for using regular expressions to solve practical real-world event-processing problems, illustrated by several dozen short EventScript examples. We explain EventScript language features as they arise in our examples, but the reader is referred to [5] for a comprehensive discussion of language constructs and rules.

This paper is structured as follows: Section 2 compares our approach to previous approaches for defining compound-event pat-

terns. In Section 3 we define our fundamental notions of an event and an event-processing program. Section 4, which constitutes the bulk of the paper, illustrates the use of regular expressions for applying ten paradigms that are common in event-processing applications; each paradigm is illustrated by an example involving embedded devices, an example involving cyber-physical systems, and an example involving business. Section 5 observes that EventScript programming is conducive to distinct styles of programming, in which the state of the computation is reflected either by current positions within a regular expression or by the values of variables. Section 6 describes current implementations of EventScript and Section 7 summarizes our argument for the use of regular expressions.

2. PREVIOUS APPROACHES

Most approaches to the specification of event patterns can be traced back to active databases, such as HiPAC [7]. Event-condition-action rules for such databases can be written at a more abstract level by viewing certain combinations of low-level database events as constituting higher-level *compound events*.

In the COMPOSE event system [9] for the Ode object database, a compound event $E[h]$ is a subset of a *history*, h, of event occurrences. Compound-event operators are defined in terms of set operations on event histories. A primitive event p maps a history h to the set of all event occurrences of p in h; $E \wedge F$, where E and F are event expressions, maps h to $E[h] \cap F[h]$; $!E$ maps h to $h - E[h]$. Other compound-event operators are defined in terms of these. For example, *prior*(E,F) takes place when F takes place and E has taken place some time earlier; *sequence*(E,F) takes place when E takes place on one event occurrence and F takes place on the next; $(<n>E)$ takes place the n^{th} time that E takes place; (*every* $<n>$ E) takes place every n^{th} time that E takes place; $E \mid F$ maps a history h to $F[E[h]]$. Named composite events are defined by rules, for example $A(x,y) = prior(B(x),C(y))$, that associate a name with an event expression and specify how attributes of the composite event are computed.

The NAOS compound-event model [6] features event expressions consisting of placeholders for single primitive events of a specified type, joined by eight operators: negation, disjunction, conjunction, exclusive or, sequence (in which the set of primitive events matching the left operand may overlap in time with the set matching the right operand), strict sequence (in which all primitive events matching the left operand must precede all events matching the right operand), iteration (matching a fixed number of occurrences of events that each match the operand, with the set of primitive events matching the operand on different iterations potentially overlapping in time), and strict iteration (matching a fixed number of occurrences of events matching the operand, with the set of primitive events matching the operand on one iteration preceding the set of primitive events matching the operand on the next iteration). Negation and exclusive or are defined in part in terms of the absence of an event during a validity interval. Validity intervals of expressions are defined recursively in terms of the subexpression structure.

In SAMOS [8], compound events are constructed from other events using sequence, conjunction, disjunction, testing for the absence of an event during a specified *monitoring interval*, and collapsing occurrences of a specified type of event during a specified monitoring interval into a single occurrence. A monitoring interval is defined in terms of starting and ending events or times, or by the union, intersection, or repetition of other monitoring intervals. Fixed rules define the attributes of a compound event in terms of the attributes of its constituents. A *coupling mode* determines whether the condition of an even-condition-action rule and any resulting action are processed immediately upon event detection, deferred until the end of the triggering transaction, or executed asynchronously. Programmed priorities determine the order in which multiple eligible rules are executed.

In Amit [1], a rule defining an event pattern is active during a *lifespan* delimited by events or times. Some rules are triggered by the presence or absence of certain sets of event occurrences during the lifespan (e.g., occurrences of each of a specified list of event types, optionally in a specified order; any single occurrence of any of a specified list of event types; the occurrence of at least a specified number of such events, optionally with the stipulation that each event be of a different type; the occurrence of at most a specified number of events of one of a specified set of event types, optionally counting at most one event of each type; or, for two specified event types, an occurrence of the first type and no occurrence of the second type). Other rules are triggered by the passage of time. Each event type in a rule may be accompanied by a predicate that an arriving event must satisfy; a predicate indicating whether the arriving event should contribute to multiple situation occurrences or just one; and a *quantifier* indicating whether to use all eligible events of that type that arrive during the lifespan, only the first, or only the last. There is an option to control whether an event that cannot contribute to a situation as soon as it arrives should be discarded or retained for possible use later. For situations that can be detected before the end of the lifespan, there is an option to timestamp the situation with either the time of detection or the time of lifespan termination. Amit also provides an elaborate set of options for managing lifespan initiation and termination.

In the Rapide event pattern language [12], an input adapter orders events in a partial order called *causal order*, in a manner not specified by Rapide. A *basic event pattern* matches a set consisting of a single event occurrence, with suitable attribute values. More complex patterns consist of two patterns joined by a binary *relational operator*. Three of the relational operators, called *structural operators*, match all unions $P \cup Q$ of event sets such that for all $p \in P$ and $q \in Q$, p and q have a particular temporal or causal relationship. Other relational operators are defined in terms of traditional set operations. A *repetition pattern* specifies a number of repetitions (possibly unbounded), a structural operator, and a subpattern to be repeated, with the stipulation that the structural operator hold between the event sets matched on subsequent iterations. Any pattern may have a predicate that must be true in order for the pattern to match.

Other mechanisms for specifying compound events include the Snoop event-specification language for the Sentinel database [3] and Trigs [11].

While many previous approaches introduce mathematical notions and abstractions unfamiliar to the typical programmer, regular expressions are familiar to the users of languages like awk, C#, Java, JavaScript Perl, PHP, Python, Ruby, and Visual Basic and tools like grep, sed, and vi. Unlike subsets of event histories, va-

lidity intervals, monitoring intervals, rule priorities coupling modes, lifespans, quantifiers, and matching event sets, regular expressions offer an intuitive underlying model—the matching of a sequence of symbols against a pattern that can be reduced to sequence, repetition, and alternative constructs.

3. THE EVENT MODEL

Before presenting examples, we explain our fundamental view of what an event is and what an EventScript program does. An EventScript program consumes input events that arrive in some well-defined order and emits output events in a well-defined order. (Events emitted by one EventScript program may be consumed by another.) Every event has a name and carries a value belonging to some data type. All events with a given name carry values of the same data type. EventScript has primitive data types such as `boolean`, `long`, `double`, `string`, and `time`, as well as structure types and array types. The correspondence between event names and entities in the real world is not defined by EventScript, making the language applicable in a wide variety of milieus.

4. BASIC PROGRAMMING PARADIGMS

In this section, we consider ten paradigms that arise repeatedly in event-processing applications, and show how regular expressions can be used to apply those paradigms. The ten paradigms are:

- event translation
- monitoring during an interval
- periodic processing
- filtering
- reacting to the absence of an event
- joining data from asynchronous streams
- event grouping
- associative lookup
- piping

We illustrate each paradigm with an example involving embedded devices, an example involving a cyber-physical system (i.e., one performing data processing related to the physical world), and a business example. The line between device-level and cyber-physical applications, and the line between cyber-physical and business applications, are fuzzy. We have not agonized over the appropriate classification of borderline examples, but have endeavored to present a collection of examples that span the spectrum from clearly device-level to clearly business-level.

4.1 Event Translation

Perhaps the simplest kind of event-processing application is one that *translates* each event in an input stream into a corresponding output event. The translation may involve, for example, reformatting, conversion of units, or the dropping of certain attributes.

4.1.1 Device-Level Example

A sensor periodically emits a scaled integer representing 1000 times the sensed temperature in degrees Celsius, but our application logic requires temperatures expressed in degrees, as floating-point numbers. The following EventScript program implements a "virtual sensor" of the kind required by the application, taking events of the form produced by the hardware as input and emitting events of the form required by the application logic as output:

```
in long RawReading
out double DegreesC

( RawReading(n) { !>DegreesC(n/1000.0); } )*
```

The first two lines indicate that there is an input event named `RawReading`, carrying a value of type `long`, and an output event named `DegreesC`, carrying a value of type `double`. These event declarations are followed by a regular expression of the form R^*, denoting a sequence of zero or more occurrences of event sequences matching the pattern R. In this case, the pattern R is a sequence consisting of the *event marker* `RawReading(n)` followed by the action sequence `{ !>DegreesC(n/1000.0); }`. The event marker matches a single `RawReading` input event and assigns the value carried by that event to the variable `n`. (The target variable in an event marker can be omitted if there is no need to capture the value carried by the matched event.) The action sequence consists of a single action, which computes the value `n/1000.0` and emits a `DegreesC` event carrying that value.

4.1.2 Cyber-Physical Example

A system needs to translate incoming `DegreesC` events into `Cold`, `Normal`, and `Hot` events, depending on whether they carry temperatures less than 5°C, in the range 5°C to 25°C, or greater than 25°C, respectively. Here is the program:

```
in double DegreesC
out {} Cold, {} Normal, {} Hot

( DegreesC(t)
  {   t < 5.0 ? !>Cold({});
    : t <= 25.0 ? !>Normal({});
    : !>Hot({});
  }
)*
```

The output events `Cold`, `Normal`, and `Hot` are declared to carry values of type `{}`, the structure type with no fields; this is the type of value carried by events that serve as pure signals, carrying no information other than their names. The action in this case is a conditional action that executes one of three different emit actions depending on the value of `t`.

Another solution to this problem uses an EventScript feature called *event classification*. In the program

```
in double DegreesC
      case {   DegreesC < 5.0 ? ColdIn
             : DegreesC <= 25.0 ? NormalIn
             : HotIn }

out {} Cold, {} Normal, {} Hot

(   ColdIn() { !>Cold({}); }
  |
    NormalIn() { !>Normal({}); }
  |
    HotIn() { !>Hot({}) }
)*
```

the `case` construct in the declaration of input event `DegreesC` specifies that each incoming `DegreesC` event will be *classified* as either a `ColdIn`, `NormalIn`, or `HotIn` event, depending on the value it carries, and matched accordingly. This program contains a regular expression of the form R_1 `|` ... `|` R_n, which matches any event sequence matching any of the regular expressions R_1 ... R_n. R_1 ... R_n are called *alternatives*. Each alternative in this example

includes an emit action that is executed when it is reached in the matching process.

4.1.3 Business example

Incoming `Sale` and `Purchase` events with unsigned cash amounts are to be translated into outgoing `BalanceChange` events with signed cash amounts:

```
in {string saleID; long amount;} Sale,
   {string purchaseID; long amount;} Purchase

out {string transationType;
     string transactionID;
     long amount;} BalanceChange


(   Sale(s)
    { !>BalanceChange
           ( {transactionType: "S",
              transactionID: s.saleID,
              amount: s.amount} );
    }
  |
    Purchase(p)
    { !>BalanceChange
           ( {transactionType: "P",
              transactionID: p.purchaseID,
              amount: p.amount} );
    }
)*
```

The input and output events of this program carry values belonging to structure types, which have fields with specified names and values. The expressions in the emit actions specify the construction of structure values, given expressions for each field of the structure.

4.2 Monitoring During an Interval

The notion of reacting to an event that occurs during an interval of interest is so fundamental to event processing that some event-pattern formalisms have special constructs to define such intervals. In EventScript, the events marking the beginning and end of the interval are simply included in the pattern to be matched.

4.2.1 Device-Level Example

The following program runs a radiation detector that samples the number of particles striking it during the first ten seconds of every minute and emits a Sample event at the end of each sampling interval:

```
in { } Particle
out {time sampleTime; long count; } Sample

( Particle()*
  arrive[.:.:00](startOfMinute) { n=0; }
  ( Particle() {n=n+1;} )*
  arrive[.:.:10]()
  { !>Sample
        ({sampleTime:startOfMinute, count:n}); }
)*
```

In addition to *named events* corresponding to event messages arriving from the outside, EventScript recognizes *time events*, corresponding to the arrival of a particular time. Such an event can be matched by an event marker of the following form:

`arrive[`*year*`-`*month*`-`*day* *hour*`:`*minute*`:`*second*`:`*msec*`:`*µsec*`](`*t*`)`

Any of the fields may be replaced by a wildcard, denoted by a dot. If the matching process is positioned just before this event marker when a matching time is reached, the event marker is matched and the time of matching (a value of type `time`) is stored in the variable *t*. Certain fields can be omitted, with natural defaults. Thus this program matches repetitions of a pattern consisting of zero or more `Particle` events that are not counted, the arrival of second 0 of any minute, zero or more `Particle` events that are counted, and the arrival of second 10 of any minute.

4.2.2 Cyber-Physical Example

The beginning and end of the interval of interest need not be time-related. Consider a home-security system that includes a motion detector. The system's control processor receives an `Activate` event when the alarm is activated, a `Deactivate` event when the alarm is deactivated, and a `Motion` event each time the system detects motion. A `Motion` event arriving during an interval in which the alarm is activated should cause an `IntrusionAlert` event to be emitted, but a `Motion` event received at any other time should be ignored. The regular expression in the following program describes the life cycle of this system:

```
in { } Activate, { } Deactivate, { } Motion
out { } IntrusionAlert

( Motion()*  //ignore
  Activate()
  ( Motion() { !>IntrusionAlert(); } )*
  Deactivate()
)*
```

4.2.3 Business Example

By computing time values, an EventScript programmer can construct an interval of interest that starts with a named event and ends a specified amount of time after the arrival of that event. For example, the following program enables a business process to issue an `Audit` event if two or more `SteelDelivery` events occur within a 24-hour period:

```
in string SteelDelivery
out string Audit

( SteelDelivery()
  { endOfInterval=hoursAfter(24,now()); }
  ( SteelDelivery(deliveryInfo)
    {!>Audit(deliveryInfo);}
  )*
  arrive[(endOfInterval)]()
)*
```

(The action setting the variable `endOfInterval` calls the built-in functions `now`, which returns the current time, and `hoursAfter`, which returns the time a specified number of hours after a specified time. In an `arrive` event marker, the date-time pattern can be replaced by a parenthesized expression of type `time`.)

4.3 Periodic Processing

Time events can be used to achieve periodic behavior.

4.3.1 Device-Level Example

The following program receives `DeviceReading` events at arbitrary intervals and issues `PeriodicReport` events, carrying the value of the most recent `DeviceReading` event, every second:

```
in double DeviceReading
out double PeriodicReport

{ nextReport=secondsAfter(1,now());
  latestValue = 0.0; }

(   DeviceReading(latestValue)
  |
    arrive[(nextDeadline)]()
    { !>PeriodicReport(latestValue);
      nextReport=secondsAfter(nextReport,1); }
)*
```

The timing of input events and the timing of output events are decoupled: If several input events arrive during the same second, only the last one is reported; if no input events arrive during some second, the latest one from some previous second is reported.

4.3.2 Cyber-Physical Example

The following program counts cars entering and leaving a parking garage, and reports once each minute on the number of cars in the garage:

```
in { } CarEntering, { } CarLeaving
out long CarCount

{ count=0; }

(   CarEntering() { count=count+1; }
  |
    CarLeaving() { count=count-1; }
  |
    arrive[.:.]() { !>CarCount(count); }
)*
```

4.3.3 Business Example

The following program reports total sales for the day at 5:00 p.m. each day:

```
in double SaleAmount
out double DailySalesTotal

( { sum = 0.0; }
  ( SaleAmount(x) {sum=sum+x;} )*
  arrive[17:00]()
  { !>DailySalesTotal(dailyTotal); }
)*
```

4.4 Filtering

Filtering is the emitting of an output stream containing events that correspond to a subset of the events in an input stream.

4.4.1 Device-Level Example

The following EventScript program presents a sensor that emits repeated temperature readings as a virtual device that emits alerts about temperature readings over 150°C:

```
in double Temperature
   case { Temperature > 150.0 ? HighTemperature }

out double OverheatingAlert

( HighTemperature(t)
  { !> OverheatingAlert(t); }
)*
```

(The “else part” of an event-classification `case` construct can be omitted. An input event not satisfying any of the conditions in the `case` construct is dropped from the sequence of events that is matched against the regular expression.)

4.4.2 Cyber-Physical Example

A common form of stateful filtering is duplicate filtering. The following program in a cyber-physical system receives a continual stream of input events reporting the current zone in which a piece of equipment is located, and emits a single output event only when the current zone changes:

```
in string CurrentZone
out string NewZone

{ previousZone=""; }

( CurrentZone(z);
    { z != previousZone ?
          { !>NewZone(z); previousZone=z; }
    }
)*
```

4.4.3 Business Example

Another form of stateful filtering is reporting when certain thresholds have been crossed. The following program receives a `MarketOpen` event at the start of a trading day, a `MarketClose` event at the end of the trading day, and a stream of `Ticker` events during the trading day. The program emits `Up` and `Down` events when a quote for stock XYZ has risen or fallen two percent from the first trade of the day or from the amount reported in the most recent `Up` or `Down` event of the day:

```
type StockQuote = {string symbol; double price; }

in { } MarketOpen,
   { } MarketClose,
   StockQuote Ticker
      case {Ticker.symbol="XYZ" ? XYZQuote}

out StockQuote Up, StockQuote Down

( MarketOpen()
  XYZQuote(q)  {thresholdsNeedSetting = true; }

  (  { thresholdsNeedSetting ?
         { topThreshold = 1.02*q.price;
           bottomThreshold = .98 =*q.price;
           thresholdsNeedSetting = false; }
     }
     XYZQuote(q)
     {  q.price >= topThreshold ?
           {!>Up(q); thresholdsNeedSetting=true;}
      : q.price <= bottomThreshold ?
          {!>Down(q); thresholdsNeedSetting=true;}
     }
  )*

  MarketClose()
)*
```

4.5 Reacting to the Absence of Events

Event processing often entails reacting to the fact that an event has *not* occurred within some interval of interest.

4.5.1 Device-Level Example

The following program repeatedly reports when a signal from a radio beacon has not been received for 10 seconds:

```
in { } Signal
out { } Timeout

( Signal() | elapse[10 sec]() {!>Timeout({});} )*
```

4.5.2 Cyber-Physical Example

To ensure that every package on a conveyor belt has an RFID tag, we use an RFID reader alongside the belt with light-beam sensors before and after it. A package is assumed to be within range of the RFID reader from the time it interrupts the first light beam until the time it interrupts the second light beam. If no RFID reading occurs during this interval, a `MissingTag` event is emitted. The same RFID tag may be read several times as it travels down the conveyor belt, and on occasion an RFID tag might be read from a distance, before it is between the light-beam sensors. The following program checks for the absence of a `TagRead` event between a `Beam1Blocked` event and a subsequent `Beam2Blocked` event:

```
in {} Beam1Blocked, {} TagRead, {} Beam2Blocked
out { } MissingTag

( TagRead()*
 Beam1Blocked()
 (   TagRead()+ Beam2Blocked()
   |
     Beam2Blocked() { !>MissingTag({}); }
  )
)*
```

(The regular expression `TagRead()+` matches *one* or more occurrences of `TagRead` events.)

4.5.3 Business Example

The following program, enforcing a safety process, issues an `InspectionOverdue` event when three days pass without the arrival of an `InspectionCompleted` event:

```
in { } InspectionCompleted
out { } InspectionOverdue

( {deadline=daysAfter(3, now());}
  (   InspectionCompleted()
    |
      arrive[(deadline)]()
      {!>InspectionOverdue({});}
  )
)*
```

4.6 Joining Data from Asynchronous Streams

Much event processing entails receiving data arriving asynchronously from two or more input streams, maintaining some sort of current state based on the latest data from each stream, and emitting output events based on the current state. Output events might be triggered by arrival of a new event from one of the streams, arrival of a new event from any of the streams, or by the passage of time.

4.6.1 Device-Level Example

A controller sets a warning light may either to remain off, to blink once a second, or to remain on. There are many client processes that may send the warning-light controller either a `StartSoftAlert` event followed eventually by an `EndSoftAlert` event, or a `StartHardAlert` event followed eventually by an `EndHardAlert` event. The current state in this case consists of the number of pending soft alerts and the number of pending hard alerts. If there is at least one hard alert pending, the light should remain on; otherwise, if there at least one soft alert, the light should blink; otherwise, the light should remain off. Here is a program that receives events starting and ending hard and soft alerts and issues alternating `On` and `Off` events to control the light:

```
in { } StartSoftAlert, { } EndSoftAlert,
   { } StartHardAlert, { } End HardAlert

out { } On, { } Off

{ softCount=0; hardCount=0; lightOn=false;
  nextBlinkTime=millisecondsAfter(500,now()); }

(   StartSoftAlert(){ softCount=softCount+1; }
  |
    EndSoftAlert(){ softCount=softCount-1; }
  |
    StartHardAlert()
    { hardCount=hardCount+1;
      hardCount==1 && !lightOn?
         { !>On({}); lightOn=true; }
  |
    EndHardAlert()
    { hardCount=hardCount-1;
      hardCount==0 && softCount==0 ?
         { !>Off({});lightOn=false; }
    }
  |
    arrive[(nextBlinkTime)]()
    { hardCount==0 && softCount>0 ?
         { lightOn ?
               { !>Off({}); lightOn=false; }
             : { !>On({}); lightOn=true; }
         }
      nextBlinkTime =
         millisecondsAfer(500,nextBlinkTime);
    }
)*
```

4.6.2 Cyber-Physical Example

A software air-conditioning thermostat receives `SetTarget` events from a process that uses personal preferences, current regional demand for electricity, and the time of day to compute target temperatures. The thermostat also receives `Temperature` events from a temperature sensor, and emits a `TurnCoolingOn` event when the current temperature rises to two degrees above the target temperature, followed by a `TurnCoolingOff` event when the current temperature falls to two degrees below the target temperature. In this case, the current state consists of the target temperature and the actual measured temperature. Here is the EventScript program for the thermostat:

```
in double SetTarget, double Temperature
out TurnCoolingOn, TurnCoolingOff

{ target=25; actual=25; coolingOn=false; }

( ( SetTarget(target) | Temperature(actual) )
  {   actual >= target+2.0 && !coolingOn ?
        { !TurnCoolingOn({}); coolingOn=true; }
    : actual <= target-2.0 && coolingOn ?
        { !TurnCoolingOff({}); coolingOn=false; }
  }
)*
```

4.6.3 Business Example

A warehouse replenishment process receives an `OrderReceived` event when an order is received from a customer, a `ShipmentReceived` event when products are delivered to the warehouse from the manufacturer, and an `OrderFulfilled` event when items are shipped from the warehouse to the customer. Each of these events carries a number of items. When the current warehouse inventory falls more than 10 below the number of items in pending orders, the process issues a `ResupplyRequest` event to

request the manufacturer to deliver a number of items that will raise the inventory to 20 more than the number of items in pending orders. The current state consists of the number of items in the warehouse inventory and in pending orders. Here is the program:

```
in long OrderReceived, long ShipmentReceived,
   long OrderFulfilled

out long ResupplyRequest

{ pendingFulfillment=0; inStock=0;  }

(   (   OrderReceived(itemCount)
        { pendingFulfillment =
             pendingFulfillment+itemCount; }
      |
        OrderFulfilled(itemCount)
        { inStock=inStock–itemCount; }
    )
    { surplus =
        inStock – pendingFulfillment;
      surplus < 10 ?
        !>ResupplyRequest(20-surplus);
    }
  |
    ShipmentReceived(itemCount)
    { inStock=inStock+itemCount; }
)*
```

4.7 Event Grouping

It is often useful to group arriving events based on the values they carry, and to look for patterns only among events in the same group.

4.7.1 Device-Level Example

We wish to report closely spaced readings of the same RFID tag by the same reader as a single event. In effect, we want to perform duplicate elimination separately for each combination of tag and reader. The following program reports readings of the same tag by the same reader as a single event unless they are separated by a ten-second interval in which that tag was not read by that reader:

```
type RFIDReading =
    {string readerID;
     string tagID;
     time timestamp;}

in  RFIDReading RawReading
   group(RawReading.readerID, RawReading.tagID)

out RFIDReading UniqueReading

RawReading(r) { !>UniqueReading(r); }
RawReading()*
elapse[10 seconds]()
```

The `group` clause in the declaration of `RawReading` stipulates that incoming `RawReading` events will be grouped according to a *grouping key* computed from the value carried by each event. In this case, the grouping key has two parts, corresponding to the `readerID` and `tagID` fields of the incoming `RawReading` event. (The values of grouping-key parts may be specified by arbitrarily complex expressions; within these expressions, the name of the event being declared—`RawReading` in this case—represents the value carried by the incoming event for which a grouping key is to be computed.) In effect, for each grouping-key value, a separate instance of the program executes, and each incoming event is directed to the instance corresponding to its grouping key.

In this example, each grouping-key value corresponds to a unique reader/tag combination. The first reading for a given combination is echoed in a `UniqueReading` output event. Subsequent `RawReading` events for the same reader and tag match `RawReading()*`, and are ignored until 10 seconds elapse without such an event. Then the `elapse` event marker is matched, and the execution instance corresponding to this grouping key terminates.

4.7.2 Cyber-Physical Example

An active-badge location-tracking system enforces a rule that a visitor to a business and his host or must remain within 10 meters of each other. The system issues an alert when the host and visitor are too far apart, or when a minute passes without a position update for one of them. Locations within the building are described by a two-dimensional coordinate system in which one unit equals one meter. When a visitor enters the building, at location (0,0), he is issued a badge by a receptionist, who registers the host assigned to that visitor. The location-tracking system then begins issuing a stream of `Visitor` events, carrying the visitor's ID and location. It also tracks the location of each registered host, and issues a `Host` event for each visitor assigned to that host, containing the ID of the *visitor* and the location of the host. This allows an EventScript program to group `Visitor` and `Host` events by visitor ID, so that there is a separate execution instance for each visitor, tracking the location of that visitor and his host. When the visitor turns in his badge at the end of the visit, an `Unregister` event containing the visitor's ID is sent to the EventScript program to signal that the execution instance corresponding to that visitor ID can be terminated. Here is the program:

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

type BadgeReport =
   {string visitorID; Point location;}

in BadgeReport Host group(Host.visitorID),
   BadgeReport Visitor group(Visitor.visitorID),
   string Unregister group(Unregister)

out BadgeReport UnaccompaniedVisitor,
    string HostNotSeen,
    string VisitorNotSeen

{ INITIAL_REPORT =
    { visitorID: group[0],
      location: {x: 0.0, y: 0.0} };
  h = INITIAL_REPORT;
  v = INITIAL_REPORT;
  hostDeadline = minutesAfter(1, now());
  visitorDeadline = hostDeadline;
}

(   (   Host(h)
        {hostDeadline=minutesAfter(1,now());}
      |
        Visitor (v)
        {visitorDeadline=minutesAfter(1,now());}
    )
    { deltaX = v.location.x-h.location.x;
      deltaY = v.location.y-h.location.y;
      deltaX*deltaX + deltaY*deltaY > 100 ?
         !>UnaccompaniedVisitor
            ( {visitorID: group[0],
               location: v.location} );
    }
  |
```

```
    arrive[(hostDeadline)]()
    { !>HostNotSeen(group[0]);
      hostDeadline =
         minutesAfter(1,hostDeadline); }
  |
    arrive[(visitorDeadline)]()
    { !>VisitorNotSeen(group[0]);
      visitorDeadline =
         minutesAfter(1,visitorDeadline); }
)*
Unregister()
```

For each execution instance of a program that uses grouping, an expression of the form `group[n]` evaluates to the n^{th} part of that instance's grouping key, where parts are numbered starting from zero. (In this example the grouping key has only one part.) In addition to grouping, this program illustrates several paradigms that we discussed earlier: reacting to the absence of events, periodic processing, and the joining of data from asynchronous streams.

4.7.3 Business Example

The warehouse replenishment example of Section 4.6.3 assumes that there is only one kind of item to be tracked. We can easily generalize this program by adding an item ID to each event and grouping by item ID:

```
type ItemInfo{string itemID; long count;}

in ItemInfo OrderReceived
              group(OrderReceived.itemID),
   ItemInfo ShipmentReceived
              group(ShipmentReceived.itemID),
   ItemInfo OrderFulfilled
              group(OrderFulfilled.itemID)

out ItemInfo ResupplyRequest

{ pendingFulfillment=0; inStock=0; }

(   (   OrderReceived(r)
        { pendingFulfillment =
             pendingFulfillment+r.count; }
      |
        OrderFulfilled(f)
        { inStock=inStock–f.count; }
    )
    { surplus =
        inStock – pendingFulfillment;
      surplus < 10 ?
         !>ResupplyRequest
              ( {itemID: group[0],
                 count:20-surplus} );
    }
  |
    ShipmentReceived(sr)
    { inStock=inStock+sr.count; }
)*
```

Each execution instance of this version has its own copies of variables `pendingFulfillment` and `inStock`, and, for one particular item ID, mimics the behavior of the version in Section 4.6.3.

4.8 Associative Lookup

Grouping can be used to perform associative lookup. In effect, a program with grouping implements a mapping from grouping-key values to the variables and other state information of a particular execution instance.

4.8.1 Device-Level Example

An RFID reader emits events consisting of a reader ID (a cryptic number such as a MAC address), a tag ID, and a timestamp. The following EventScript program implements an abstract RFID reader that emits events consisting of an abstract location name, a tag ID, and a timestamp:

```
in {long readerID; string locationName}
    RegisterReaderLocation
       group(RegisterReaderLocation.readerID),

    {long readerID; string tagID; time when;}
    RawReading group(RawReading.readerID)

out { string locationName;
      string tagID;
      time when; } AbstractReading

RegisterReaderLocation(registration)
{ myLocationName = registration.locationName; }

( RawReading(r)
  { !>AbstractReading
         ( {locationName: myLocationName,
            tagID: r.tagID,
            when: r.when} ); }
)*
```

At system startup, initialization code emits a `RegisterReaderLocation` event for each RFID reader, specifying the hardware reader ID and abstract location name for that reader. EventScript creates an execution instance for each hardware reader ID and sends the event to that instance, which saves the abstract location name in its own copy of the variable `myLocationName`. EventScript sends each subsequent `RawReading` event to the execution instance storing the appropriate abstract location name, so that the appropriate event translation can be performed.

4.8.2 Cyber-Physical Example

New York State has introduced a system that uses RFID tags in cars, normally used to pay tolls, to sample the travel time between two readers and post travel advisories about expected travel times **Error! Reference source not found.**[13]. The following program receives events reporting RFID readings and emits events containing anonymous samples of the travel time for a given tag between two consecutive readers:

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

out { string fromReader;
      string toReader;
      long travelTime;} Sample

Reading(r)
{ prevReader=r.readerID; prevTime=r.when; }

( Reading(r)
  { !>Sample
       ( { fromReader: prevReader,
           toReader: r.readerID,
           travelTime: minutesBetween
                         (prevTime, r.when)
         } );
    prevReader=r.readerID;
    prevTime=r.when;
  }
)*
elapse[1 hour]()
```

(The built-in function `minutesBetween` takes two values of type `time` and returns the number of minutes between those two times as a rounded integer.) The initial reading of a given tag creates an execution instance for that tag ID, and the relevant historical information for that tag is stored in the `prevReader` and `prevTime` variables of that instance. Subsequent `Reading` events for that tag are directed to the same execution instance. After an hour passes without a reading from a given tag, the corresponding execution instance matches the `elapse` event marker and the instance terminates.

4.8.3 Business Example

The following program receives `ATMUse` events whenever an ATM card is used, groups these events by card ID, and issues an alert whenever the same card is used more than once within six hours:

```
in {string cardID; string location} ATMUse
   group(ATMUse.cardID)
out string TwiceInSixHours

ATMUse()
( ATMUse() { !>TwiceInSixHours(group[0]); } )*
elapse[6 hours]()
```

The first use of a card after six hours creates a new execution instance, and subsequent uses of the card without a six-hour gap are directed to the same instance. After six hours pass without another use of the card, the `elapse` event marker is matched and execution terminates. In this case, the relevant information retrieved by associative lookup is not the value of a variable, but the instance's internal record of how much time has passed since its last `ATMUse` event.

4.9 Piping

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 the use of event grouping to group data in different ways at different stages. 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.

4.9.1 Device-Level Example

The warning-light-controller program of Section 4.6.1 addresses two distinct concerns—determining what kind of alert, if any, should be signaled and controlling the blinking of the light for soft alerts. We can separate these concerns into two simpler EventScript programs, or *stages*, by introducing intermediate events `OffMode`, `BlinkingMode`, and `OnMode` emitted by the first stage and received by the second stage. The first stage is responsible for determining when mode changes should take place, and signaling those changes:

```
in {} StartSoftAlert, {} EndSoftAlert,
   {} StartHardAlert, {} End HardAlert

out {} OffMode, {} BlinkingMode, {} OnMode

{ softCount=0; hardCount=0; }

(   StartSoftAlert()
    { softCount=softCount+1;
      hardCount==0 & softCount==1 ?
         !>BlinkingMode({});
    }
  |
    EndSoftAlert()
    { softCount=softCount-1;
      softCount==0 && hardCount==0 ?
         !>OffMode({});
    }
  |
    StartHardAlert()
    { hardCount=hardCount+1;
      hardCount==1 ? !>OnMode({});
    }
  |

    EndHardAlert()
    { hardCount=hardCount-1;
      hardCount==0 ?
         { softCount==0 ?
               !>OffMode({});
             : !>BlinkingMode({});
    }
)*
```

The second stage is responsible for emitting `On` and `Off` events in accordance with the current mode:

```
in {} OffMode, {} BlinkingMode, {} OnMode

out { } On, { } Off

{ lightOn=false; }

(   OffMode()
    { lightOn ? { !>Off({});lightOn=false; } }
  |
    OnMode()
      { !lightOn ? { !>On({});lightOn=true; } }
  |

    BlinkMode()
    { nextBlinkTime =
          millisecondsAfter(500,now()); }
    ( { lightOn ?
            { !>Off({}); lightOn=false; }
          : { !>On({}); lightOn=true; }
      }
      arrive[(nextBlinkTime)]()
      { nextBlinkTime =
         millisecondsAfer(500,nextBlinkTime);}
    )*
)*
```

The first stage does not deal with time events at all. The second stage does not deal with alert counts at all, and is easily structured so that time events occur only in blinking mode.

4.9.2 Cyber-Physical Example

The travel-time sampling program of Section 4.8.2 is actually the first stage of a pipeline whose second stage averages samples for a given pair of readers to estimate the travel time from the first reader to the second. Here is the second stage, which computes an exponential moving average:

```
in { string fromReader;
     string toReader;
     long travelTime;} Sample
   group(Sample.fromReader, Sample.toReader)
```

```
out { string fromReader;
      string toReader;
      double travelTime;} Average

{ ALPHA = 0.1;  // smoothing factor
  ONE_MINUS_ALPHA = 1.0-ALPHA; }

Sample(s) { history = double(s.travelTime); }
( { !>Average
        ( { fromReader: s.fromReader,
            toReader: s.toReader,
            travelTime: history } ) ;
    history =
       ALPHA*s.travelTime +
       ONE_MINUS_ALPHA*history;
  }
  Sample(s)
)*
```

Thus the first stage groups RFID-reading data by tag ID to produce events related to a given tag having various starting and ending readers; the second stage groups events by starting and ending reader IDs to perform a computation related to a given reader pair, using data originating from various tags. Piping in conjunction with grouping is a powerful way to cross-section a collection of data in multiple dimensions.

4.9.3 Business Example

Piping can also be used to reconcile what M.A. Jackson [10] calls a *boundary clash*. Suppose a business receives events containing various numbers of customer leads, and wishes to assign customer leads to sales staff in groups of ten. 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 an EventScript solution, two stages of a pipeline can play the role of these coroutines. A program that receives events with arbitrarily sized bundles of leads and emits events with bundles of ten leads can be written easily in two stages. The first stage disassembles incoming events and emits output events containing one lead at a time:

```
in string[] IncomingBundle
out string CustomerLead

( IncomingBundle(b)
  { for (i: 0, length(b)-1 )
       !>CustomerLead(b[i]); }
)*
```

(The EventScript type `string[]` consists of arrays with elements of type `string`, indexed starting at zero. The built-in function `length` reports the number of elements in an array.) The second stage receives events containing individual leads and assembles them into bundles of ten:

```
in string CustomerLead
out string[] OutgoingBundle

{ buffer = new string[10]; cursor = 0; }

( CustomerLead(buffer[cursor])
  { cursor==9 ?
      { !>OutgoingBundle(buffer);
        buffer = new string[10];
        cursor=0; }
    : cursor=cursor+1;
  }
)*
```

(The expression `new string[10]` allocates a new array with 10 uninitialized array elements.)

5. SYNTAX- AND DATA-DRIVEN STYLES

Regular expressions with fundamentally different structures can describe the same set of input sequences. For example, the following regular expressions all match zero or more repetitions of subsequences each consisting of either an `A` event or of a `B` event followed by a `C` event:

- `( A() | B() C() )*`
- `( A()* B() C() )*`
- `( ( B() C() )* A() )*`

Furthermore, it is possible to write a regular expression that matches a superset of the sequences we expect to encounter in an application, and to use conditional actions if necessary to ensure that sequences of no interest are properly ignored.

In the regular expressions of automata theory, the state of an execution is captured entirely in the identity of the current state in the corresponding finite automaton, or equivalently, the set of possible current positions within the regular expression. In an EventScript program, part of the current state of an execution may also be captured in variables. For example, consider a program to control a traffic light at the intersection of a north-south road and an east-west road. The light must change no more frequently than once every 30 seconds and no less frequently than once every 120 seconds. However, after 30 seconds have passed since the light last changed, it changes again as soon as a car is detected on the road that has a red light. The input events `NSCar` and `EWCar` correspond to a car being detected on the north-south road or the east-west road, respectively. The output events `NSGreen` and `EWGreen` direct the light to change so that it is green along the north-south road or the east-west road, respectively. In the following program, the state of the execution is captured primarily in the current position within the regular expression, as evidenced by the rich comments we can interleave at various points within the regular expression:

```
in {} NSCar, {} EWCar
out {} NSGreen, {} EWGreen

( { !>NSGreen({});
    currentTime = now();
    minChangeTime =
       secondsAfter(30,currentTime);
    maxChangeTime =
       secondsAfter(120,currentTime);
  }

  // The light has been green for the N-S road
  // for less than 30 sec, and must not change.

  ( NSCar() | EWCar() )*
  arrive[(minChangeTime)]()

  // The light has been green for the N-S road
  // for at least 30 sec, and must change when
  // an E-W car is detected or 120 sec have
  // passed since the last change.
```

```
  NSCar()*
  ( EWCar() | arrive[(maxChangeTime)]() )
  { !>EWGreen({});
    currentTime = now();
    minChangeTime =
       secondsAfter(30,currentTime);
    maxChangeTime =
       secondsAfter(120, currentTime);
  }

  // The light has been green for the E-W road
  // for less than 30 sec, and must not change.

  ( NSCar() | EWCar() )*
  arrive[(minChangeTime)]()

  // The light has been green for the E-W road
  // for at least 30 sec, and must change when
  // a N-S car is detected or 120 sec have
  // passed since the last change.

  EWCar()*
  ( NSCar() | arrive[(maxChangeTime)]() )
)*
```

In contrast, the following version of the program has only one current position while it is waiting for an event—at the start of the set of alternatives—and saves its state in variables:

```
in {} NSCar, {} EWCar
out {} NSGreen, {} EWGreen

{ pendingChange = "NSGreen"; }

( { pendingChange != "none" ?
       { pendingChange=="NSGreen" ?
              { !>NSGreen({});
                currentGreen = "NS"; }
            : { !>EWGreen({});
                currentGreen = "EW"; }
         currentTime = now();
         minChangeTime =
            secondsAfter(30,currentTime);
         maxChangeTime =
            secondsAfter(120,currentTime);
         nextMilestone = minChangeTime;
         changeAllowed = false;
         pendingChange = "none"; }
  }

  (   NSCar()
      { currentGreen=="EW" && changeAllowed ?
           pendingChange = "NSGreen";
      }
    |
      EWCar()
      { currentGreen=="NS" && changeAllowed ?
           pendingChange = "EWGreen";
      }

    |
      arrive[(nextMilestone)]()
      { changeAllowed ?
             { // 120 sec passed
               currentGreen="EW" ?
                    pendingChange = "NSGreen";
                  : pendingChange = "EWGreen"; }
           : { // 30 sec passed
               changeAllowed=true;
               nextMilestone = maxChangeTime; }

  )
)*
```

This program initially marks a change to N-S green as pending then enters a loop that repeatedly applies any pending change, waits for whichever event occurs next, and reacts to that event (possibly by marking a new change as pending).

There is a spectrum of programming styles between the purely syntax-driven style, in which no variables are used, and the purely data-driven style, in which there is only one current position in the regular expression. A regular expression in the syntax-driven style models the structure of the incoming event stream and has a simple state corresponding to locations in the program. It follows that various points in the regular expression can be annotated with simple problem-oriented invariants. A regular expression in the data-driven style is not all that different from writing a single event-handling routine for each possible input event. It provides confidence that all possible event sequences are handled. It is conducive to writing a single program-wide representation invariant concerned with relationships among the variables inside the program. The state machines describing some problems are difficult to express in the syntax-driven style because their state-transition graphs do not resemble the flow graphs of structured programs, but any reactive program is easily expressed in the data-driven style. We speculate that programmers with different mental models of programming might prefer different styles.

6. THE STATUS OF EVENTSCRIPT

EventScript has been implemented, and used in large programming exercises.

We have a standalone version that executes in a state-machine driver written in Java, receiving input events from an input adapter and sending output events to an output adapter. The standalone version also features a testing tool that allows a program to be executed in simulated time, obtaining input events from a time-stamped event trace. Performance tests detailed in [5] show that this implementation is capable of handling hundreds of thousands of events per second.

We have also embedded EventScript in two programming environments—DRIVE [4] and System S [2]—that support the construction of large event-processing and stream-processing programs through the use of event channels to connect input and output ports of event-processing nodes. In each case, EventScript is one of the languages available to specify the logic of a node: Input-event names correspond to the names of input ports at which events arrive and output-event names correspond to the names of output ports through which events are emitted.

7. CONCLUSIONS

Some of the device-level examples we have seen require microsecond-scale response times. Some of the business examples we have seen require transactional treatment of events, with the state of the computation stored persistently between events to facilitate recovery from system failures. It is unlikely that one EventScript implementation can satisfy both these requirements. However, the execution model of an EventScript program is extremely simple, so the construction of both a real-time implementation and a transactional implementation is not a daunting prospect.

While a single EventScript *implementation* might not be equally applicable to embedded-device event processing and business event processing, we have shown through numerous examples that EventScript *programming paradigms* are equally applicable

across the spectrum between these domains. So are EventScript programming skills. Furthermore, in contrast to event-pattern specification approaches that have been proposed in the past, regular expressions are simple, familiar, and intuitive. People who have not seen EventScript before quickly acquire the ability to read and understand EventScript programs. Furthermore, EventScript is defined in a way that allows it to fit naturally into a wide variety of event-processing environments. For these reasons, we believe EventScript can serve as a *lingua franca* for event processing.

REFERENCES

[1] Adi, Asaf, Botzer, David, and Etzion, Opher. The situation manager component of Amit—active middleware technology. In Alon Halevy and Avigdor Gal, eds., *Next Generation Information Systems and Technologies: 5th International Workshop, NGITS 2002, Caesarea, Israel, June 24-25 2002, Proceedings. LNCS* **2382**, Springer, Berlin, 2002, 158-168.

[2] Amini, Lisa, Jain, Navendu, Sehgal, Anshul, Silber, Jeremy, and Verscheure, Olivier. Adaptive control of extreme-scale stream processing systems. *26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*, 2006, 71–77.

[3] Chakavarthy, Sharma, and Mishra, Deepak. Snoop: an expressive event specification language for active databases. Tech. report UF-CIS-TR-93-007, Dept. of Comp. and Inf. Sci., U. of Florida, Mar. 1993.

[4] Chen, H., Chou, P., Cohen, N.H., Duri, S., and Jung, C. A distributed responsive infrastructure virtualization environment for sensor and actuator applications. *IBM Systems Journal* **47**, No. 2 , to appear.

[5] Cohen, Norman H., and Kalleberg, K.T. EventScript: an event-processing language based on regular expressions with actions. *ACM SIGPLAN/SIGBED 2008 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, Tucson, Arizona, June 2008 (to appear)

[6] Collet, Christine, and Coupaye, Thierry. Primitive and Composite Events in NAOS. *Actes des 12e Journées Bases de Données Avancées*, Cassis (France), August 1996, 331–349.

[7] Dayal, U., Blaustein, B., Buchmann, A., Chakravarthy, U., Hsu, M., Ledin, R., McCarthy, D., Rosenthal, A., Sarin, S., Carey, M. J., Livny, M., and Jauhari, R. The HiPAC project: combining active databases and timing constraints. SIGMOD Rec. 17, 1 (Mar. 1988), 51-70.

[8] Dittrich, Klaus R., Fritschi, Hans, Gatziu, Stella, Geppert, Andreas, and Vaduva, Anca. SAMOS in hindsight: experiences in building an active object-oriented DBMS. Technical report 2000.05, Database Technology Research Group, University of Zurich Department of Information Technology, `ftp://ftp.ifi.unizh.ch/pub/techreports/TR-2000/ifi-2000.05.pdf`

[9] Gehani, Narain, Jagadish, H. V., and Shmueli, O. COMPOSE: a system for composite event specification and detection. In Adam, Nabil R., and Bhargava, Barat K., eds., Advanced Database Systems, *LNCS* **759**, 1994, 3–15.

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

[11] Kappel, Gerti, Rausch-Schott, Stefan, and Retschitzegger, Werner. A tour on the TriGS active database system—architecture and implementation. In Proceedings of the 1998 ACM Symposium on Applied Computing (SAC '98), Atlanta, Georgia, Feb. 27 - Mar. 1, 1998, 211-219.

[12] Luckham, David. The Rapide pattern language. In *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, Boston, 2002, chapter 8.

[13] Swedberg, Claire. RFID provides ETAs to N.Y. drivers. *RFID Journal*, October 12, 2007, http://www.rfidjournal.com/article/articleview/3673/1/1/