Rule of Evidence for Parsing Event Logs

Russell G. Almond
Florida State University

October 16, 2018

Abstract

A common need in game- and simulation-based assessments is to parse a log made up of many event records either as the assessment is running or afterwards to score this process. The end result of this processing is define a collection of *observable* outcome variables which are then used for summary scoring, task based feedback or other analytics. In evidence-centered assessment design (ECD) these observable are defined by *rules of evidence*. This manual describes the specification for a language, called *EI-Event* (evidence identification from events) which specifies rules of evidence in a JavaScript query format, similar to that used by the Mongo database.

Key words: Event Logs, JSON, Evidence Identification, Test Scoring

1 Introduction

? (?) defined a generalized model for assessment that consists of four processes:

Presentation Process (PP) Presents stimulus material to examinee and captures work product.

Evidence Identification (EIP) Extracts evidence in the form of observed outcome variables from the raw work product.

Evidence Accumulation (EAP) Combines observable outcomes from several tasks to produce statistics about examinee competencies.

Activity Selection (ASP) Assigns the next task based on current competency estimates and previous observations.

This paper focuses on the second of these, the evidence identification process. In many cases, evidence identification is almost trivially simple. For multiple choice tests it simply matches the work product (the selected answer) to the key. Many

short answer question types also have simple pattern matching keys. However, complex work products, particularly event logs from simulators, require much more complex EIP.

? (?) defined an evidence model with two parts: the statistical model or *weights of evidence*—which correspond to the EAP,—and the *rules of evidence*—which corresponds to the EIP. Although the term *rules of evidence* is a pun on the legal term, literal rules of evidence were used in HyDRIVE (?, ?), one of the exemplars used in building the original evidence-centered design framework. HyDRIVE was a simulation of maintaining the hydraulics system of the F-15 aircraft. The evidence identification system for HyDRIVE was written in Prolog, and consisted of "rules" that would fire if certain conditions were met.

For game and simulation based assessments, this rule-based approach to defining the evidence identification process works fairly well. Conceptually a rule looks something like:

WHEN *context* IF *condition* THEN *observable* = *value*.

Here *context* is the context in which the evidence is gathered; in simple systems this is the task. In simulator systems the context can be an emergent property of the simulator, with several contexts present within a single task. Section **??** explores this in more detail. The *condition* often involves querying the state of the system, this in turn requires that the state of the system be monitored. Section **??** describes the use of a state machine to track the state of the simulator.

This paper defines the EI-Event protocol for handling evidence identification. It processes a collection of events in a log files by running a collection of rules. In particular, it assumes that the game or simulator is logging to a learning record store (essentially a database) event descriptions that look something like:

Example 1 Generic Event Record, JSON format.

```
{
  app:"ecd://coe.fsu.edu/EPLS/AssessmentName",
  uid:"Student/User ID",
  timestamp:"Time at which event occurred",
  verb:"Action Keyword",
  object:"Object Keyword",
  data:{
    field1:"Value",
    field2:["list","of","values"],
    field3:{part1:"complex",part2:"object"}
  }
}
```

The format of Example **??** is JSON (java script object notation). This is list of key–value pairs with the key and value separated by a colon (:). The values can be numeric values, strings, date-time objects, arrays (using the square

brackets, []), or objects (using curly braces, {}). This event format is a simplified version of the xAPI[1] format used by Learning Locker (?, ?). The first five header fields are common to every event while the format of the data field is completely open and can contain arbitrarily complex status data.

The assumption is that the the work product the EIP receives from the PP is an ordered sequence of these event records. (This could either be directly sent from the PP to the EIP or the EIP could retrieve them on demand from a learning record store.) The EIP processes these one at a time, updating the state of the system. In the end, the EIP must send one (or more) message to the EAP stating providing values for the observables seen in a particular context. This is what the EAP uses to update the student model.

2 Tasks and Contexts

? (?) deliberately used the term *task* instead of the more familiar *item* to encourage test designers to think beyond the common multiple-choice and short-answer task types. In 2003, it was obvious that assessments using more complex simulation tasks would be an important part of the future of assessment.

? (?) had a more operational definition for *task*: a task was the grain size at which information was passed between the four processes. The PP would present a task to a student and send the work product for that task to the EIP. The EIP would process the work product and send the observable for that task to the EAP. The EAP would signal to the ASP that the task was complete, and it would select the next task to present and send that information to the PP. Depending on the application, a task could be a single item, a group of items with a common stimulus, or a more complex constructed response or simulation task.

? (?) (also ?, ?) noted that in simulation and game-based assessments, tasks do not necessarily come in clean units as in more traditional assessments. Often a task will be a subsequence of events that happen in the course of a larger simulation or game. In this case, the term task is no longer quite appropriate because several measurement contexts might occur in the course of the player completing a single game task.

2.1 Three Examples

To address the more complex situations, in this paper, the unit of movement around the four process architecture is termed a *context*. The exact definition of a context depends on the assessment application. This section provides three examples of increasing complexity.

[1]The simplification is mainly replacing the *verb* and *object* values with strings where xAPI uses more complex objects. In particular, xAPI uses a full URL to uniquely define the verb and object, as the same word could have different meanings in the context of the application. In the simplified version, the vocabulary is defined by the application, allowing the message itself to be simpler.

Computer Adaptive Test. Consider first a computer adaptive test similar to ETS's Graduate Record Exam (GRE®) as it operated around 2003. In this assessment, the *context* or *task* is equivalent to an item. The ASP selects a single item, which the PP displays, the EIP scores and the EAP updates the estimate of student ability. The ASP then tries to select a new item that maximizes information about the examinee while balancing content constraints and minimizing item exposures.

Item sets, such as a reading passage followed by several items, presented a problem for the item-as-task design. If one item was chosen from a set, then the next couple of items were constrained to also be chosen from the same set. These items could be sub-optimal for satisfying information targets or content and exposures control constraints. A better solution would be to make the task for such set-based items the item set, so an optimal item set could be chosen.

It is interesting to note that more recent version of the GRE have gone from using a single item as the task to using a testlet containing 10 items or so as the task. With this kind of task, it is easier to balance information properties, content constraint and exposure control as well as to better control context effects of items.

In this example, as in most classical assessment, the context is the same as the task—both are engineered by the test designers.

Physics Playground. The game *Physics Playground* (?, ?, ?) is a game for teaching knowledge of Newtonian physics with an embedded stealth assessment. The game is level-based: each game level is a puzzle where players must maneuver a ball to reach a target balloon. In sketching levels, players draw objects which can be used to apply forces to the ball. In manipulation levels, the players manipulate the forces on the ball through the use of sliders. As in the computer adaptive assessment, the *Physics Playground* task is engineered by the designers. In this case it is the game level.

Even in a level-based game, the task boundary is not clear. Is a task a single attempt at a game level or all attempts at a game level in a playing session? How are the boundaries of an attempt defined? What if a player restarts the level? Leaves the level and then returns? The right answer to these questions will depend on the purpose of the assessment, but unlike the computer adaptive test, there is no submit answer button to define the context boundary.

Note that in *Physics Playground*, there are multiple observed outcomes per task. The game records not only whether the level was successfully completed, but also if the player used a particular tool to solve a level. This information must be extracted from the game logs by the EIP.

The *Physics Playground* PP logs events to a Learning Locker learning store (?, ?). These events are recorded as JSON objects similar to the one shown in Example **??**. Note that this format does not have a field for task or context. It must be inferred from the events. In particular, certain events in the sequence mark the start and the end of the level. So the EIP must infer the context from the information stream and must internally keep track of where the context is.

Flight Simulator. Consider a person training to be a pilot using a flight simulator. The task in this situation might consist of flying the plane from an

origin airport to a destination airport. There might be several contexts that emerge within this single task. For example, the takeoff is one task while flying the plane after it reaches cruising altitude is another. If a thunderstorm occurs along the route that might change the context to handling the weather event, after which the simulator returns to the cruising context. Finally, the approach and the landing might also be contexts.

This example makes clear the need for moving from task to context as the unit that goes around the four process loop. Here many variables about the state of the system may be related to the definition of the context.

2.2 Context as an Emerging Status

One way to detect a context change is to keep track of the state of the simulator. When the state changes in such a way as to define a new context, then the context changes. In a complete system, the EIP needs two kinds of rules to support this. First, it needs a collection of *context rules* which defines when a context changes. Second, it needs *trigger rules* to indicate when it should send the observables for a given context to the EAP (or other listeners). At this point, these are rules in the loose sense of ? (?): requirements for the EIP. Section **??** will introduce a notation for operationally specifying those rules.

In addition to taking the role of *task* in the four process architecture, contexts play another roll in the EIP. In particular, each context has an associated set of specific evidence rules. For example, evidence rules appropriate for a manipulation level in *Physics Playground* are not necessarily appropriate for a sketching level. Tagging rules by the contexts in which they are applicable allows the EIP to filter the rule set to just those which are interesting and appropriate in the current context. Note that there is also a need here for context sets, as the list of contexts for which a particular rule is needed may change as tasks are added and removed from the assessment.

The second and third examples have a certain similarity in the work product. In both cases, the game or simulator log consists of a series of events. (Section **??** describes a proposed structure for those event records.) To process these events, often it is necessary to recreate information about the state of the system in the EIP. Section **??** describes a state machine architecture.

3 State Machine

The identification of evidence often requires fairly detailed information about what is happening in the game. Often, the best evidence is when the player performs an action which changes the state of the system. To assess such evidence, the EIP often needs to at least partially recreate the state of the system. The architecture for the EI-Event protocol, therefore starts with the idea that the EIP is a finite state machine. Each user (player or student) has an associated state object. These state objects have collections of observables, timers and

flags which change in response to events. These are described in more detail below.

Note that the state machine approach to EIP is useful even in contexts which are not simulations or games. In particular, ? (?) used a finite state machine to analyze keystroke timing data. It was necessary to parse the stream of characters recorded by the logger to associate pause events with individual words or with between word or sentence spaces. This is potentially a robust approach.

3.1 Observables

Consider the game-based assessment *Physics Playground*. As described above, the context for this application is a game level. In each game level the player attempts to move a ball to the target (balloon). The player may draw objects on the screen (sketching levels) to reach the target or manipulate physics parameters or the strength of blowers producing force (manipulation levels). For each game level, the game engine determines whether or not the game level has been passed (solved) and whether a gold or silver coin was awarded for the solution. (Efficient solutions earn gold coins, and inefficient solutions earn silver coins.)

Here are several observables identified in version 2 of *Physics Playground*:

Obs1 What is the maximum value coin (gold, silver or none) that the player has earned for this level?

Obs2 Did the player manipulate the gravity slider?

Obs3 How many objects did the player draw?

Obs4 How much time (excluding time spent on learning supports) did the player spend on the level?

Obs5 Did the player attempt to draw a springboard?

The first two are fairly simple. The state machine needs an *observable variable* which keeps track of the coin reward (Obs1) or the use of the gravity slider (Obs2). The when an appropriate event comes through the log, an evidence rule will update the appropriate variable. Obs3 is similar; in this case, the observable is a counter and it is incremented each time an event comes through indicating that an object was drawn.

The timing observable (Obs4) need a bit more work. The timer must be started when the level starts and paused every time the player starts using a learning support, resuming when the player returns from the learning support. The next section describes timers in more detail.

Detecting springboards is actually quite tricky. Obs5 requires determining if each object that the player has drawn is a springboard or not. A springboard has several elements (the springboard, the anchor holding one end fixed, and usually a weight which gives it elastic potential energy); identifying these

springboard components requires details of the object stored in the underlying physics engine. In this case, the PP of *Physics Playground* contains code to identify agents of force and motion. The PP then sends an event saying that the agent identification system identified the creation of a springboard (or other agent), which the EIP can use to build up observables.

Although generally evidence rules are placed in the EIP, in some cases it may make sense to place them in other processes. In the example above, it was easier to implement the agent identification system in the PP as it had access to the details of the screen layout, and object placement and movement. Similarly in the case of a multiple choice test, it is is probably best to have the PP send a message containing which option was selected as the event, rather than the lower level event of where the student clicked on the screen. The latter requires details of how the item is laid out in the screen that are needed in the PP but not the EIP.

There may also be cases where the EAP can handle the evidence rule. For example, consider Obs3 (which is a count) and Obs4 (which is a time). If these are to be used in a model with discrete observables (e.g., a Bayesian network) then the continuous or count variable might need to be cut into categories (e.g., low, medium, high). This might be easier to do in the EAP, especially if the Bayes net software can associate ranges of numeric values with states.

3.2 Flags and Timers

Not all potential observables need to be reported out of the EIP. In general, an observable is reported for one of three reasons:

1. It will be used by the EAP to accumulate a score for the student.
2. It will be used to customize feedback presented to the students.
3. It will be logged to a data base for future analysis (for scientific research or to improve the performance of the assessment).

Observables taking on one or more of these roles are *final observables*, as opposed to *intermediate observables* whose role is to aid in the computation of final observable values. ETS's e-rater® system (?, ?) is a good example. The system has low level code that identifies likely grammar, usage, mechanics or style errors. (These low level observables are sometimes used to give students feedback on their writing.) These low-level errors are accumulated into count variables providing the number of grammar, usage, mechanics and style errors the writer made. These count variables (after suitable normalization), along with other variables related to vocabulary and discourse, are fed into a regression model to produce the final score. Note that there are observables a number of different levels in this system: any of which could be intermediate or final depending on the requirements of the assessment.

Because EIP designers may want to separate out the intermediate and final observables, the model design has two collections of observables: the observables

collection for storing final observables and the flags collection for storing intermediate observables. However, this distinction is not strictly enforced: variables in both the flags and observable collections may play the role of intermediate observables.

Timing variable are special. Looking at the definition of Obs4, it is clear that the timer must be started at the beginning of the level, paused when the player enters the learning support and resumed when the player leaves the learning support. In order to do this, the timer object must support the following operations:

resume Continue accumulating time.

pause Stop accumulating time.

reset Set the accumulated time to zero.

(re)start Combination of reset and resume. This will also create a timer if none has been created.

value Return the time accumulated so far.

These operations can be triggered by rules. Also, the current value of a timer can be copied into an observable or flag variable. Finally, the current value of a timer can be queried in the condition part of the rule.

Thus, a state object contains:

uid An identifier for the user (student, player)

context An identifier for the current context.

oldContext An identifier for the previous context; in the event that the context has changed.

observables A collection of observable variables.

timers A collection of timers.

flags A collection of other (intermediate observable) variables.

timestamp The timestamp of the last event processed by the state.

3.3 Events

Look once more at the format of the event record in Example **??** (reproduced in Example **??**). There are five fields in the header—`app`, `uid`, `timestamp`, `verb`, and `object`—and a `data` container that can contain an arbitrary number of fields. The `data` field allows the PP to pass arbitrary information to the EIP.

The `verb` and `object` fields should be keywords appropriate to the application. These fields are taken from the xAPI (?, ?) data structure. The verb and object together should define what is happening. Some example pairs

Example 2 Generic Event Record, JSON format.

```
{
  app:"ecd://coe.fsu.edu/EPLS/AssessmentName",
  uid:"Student/User ID",
  timestamp:"Time at which event occurred",
  verb:"Action Keyword",
  object:"Object Keyword",
  data:{
    field1:"Value",
    field2:["list","of","values"],
    field3:{part1:"complex",part2:"object"}
  }
}
```

from *Physics Playground* include {("snapshot", "ball"), ("exited", "level"), ("exited", "learning support"), ("identified", "game object")}. Note that in some cases, the object is necessary to distinguish what the verb is operating on ("exited learning support" versus "exited level"). In all cases, the extra data provides additional information (the position of the ball for a snapshot event, the learning support or level exited, what the object was identified as).

The PP and the EIP need to agree on what are the valid values for the verb and object fields. This is the role of the `app` (short for application) field. Note that this is a long URL-like structure giving the name of both the specific vocabulary set and the organization that defined the vocabulary. (This is a simplification from the xAPI format, where both the `verb` and `object` values were more complex objects which provide information about the vocabulary used. In most cases, the application defines the verbs, objects and the expected data components for each verb object pair, so longer identifiers for verbs and objects are not needed.)

The remaining two header fields are straightforward. The `uid` field is matched with the state object, so that the system can track many users at the same time. The `timestamp` is just the time at which the event occurred.

3.4 Dot notation

In general, rules need to be able to reference both the state object and the event object. In particular, they need to reference specific fields in those objects. The JavaScript dot notation provides a simple mechanism for doing this. The dots define subcomponents of objects. Starting with the `state` and `event` objects, the legal fields are:

`state.context` The current context that the state object is in.

`state.oldContext` The the context of the state at the end of the previous event. In particular, this can be compared to the context to check if the

context has changed as a result of the event.

`state.observables.`*`name`* The value of the observable named *name*.

`state.timers.`*`name`* The current elapsed time of the timer named *name*.

`state.flags.`*`name`* The value of the observable named *name*.

`event.verb` The verb associated with the current event.

`event.object` The object associated with the current event.

`event.timestamp` The time at which the event occurred.

`event.data.`*`name`* The value of the extra data field named *name*.

Note that there potentially can be further dots after the *name* of the value. In particular, if the value of the observable, flag or extra data element is itself a compound object, the the dot notation can be used to refer to its fields. Also, in the predicate of rules, the dot notation applied to timers allows access to the pause, resume and reset operations.

4 Rules

Conceptually a rule has the following format: `WHEN` *`context`*`,` *`verb`*`,` *`object`* `IF` *`condition`* `THEN` *`predicate`*. The *condition* is logical expression involving the fields of the state and the current event. If this is true, then the *predicate* is executed. The *context*, *verb* and *object* are really part of the condition. Separating them into separate components allows the rule set to be indexed by those fields, which should improve the speed of execution. Example **??** shows a rule set in JSON format.

Example 3 Generic Rule, JSON format.

```
{
  ruleName:"Human readable identifier",
  doc:"Human language description",
  context:"Context Keyword, Context Group
     Keyword or ALL",
  verb:"Action Keyword" or ALL,
  object:"Object Keyword or ALL",
  ruleType:"Type Keyword",
  priority:"Numeric Value",
  condition:{...},
  predicate:{...}
}
```

The condition (Section **??**) and predicate (Section **??**) are also JSON objects. They are adapted from the query language used by the Mongo database (?, ?). The *type* and *priority* fields are used to determine the sequence in which rules are run. Sections **??** and **??** describe managing rule sets in more detail.

4.1 Types and Timing

For the most part, the predicates of the rules change the state object or have other side effects. This means that the sequence in which the rules are executed may have an effect on the final state of the system. When constructing a rule set for a given application, the designer needs to be able to provide a partial ordering on the rules: forcing the order when sequence is important.

The EI-Event protocol provides two mechanisms for sequencing rules: type and priority. The type mechanism ensures that rules that affect the state of the system are run before rules that report on the state. The priority mechanism controls sequencing within a type.

There are five types of rules, which are run in the following sequence:

1. *State Rules.* These rules should have predicates which set flag variables and manipulate timers. These rules are run first.

2. *Observable Rules.* These rules should have predicates that set observable values, they are run immediately after the state rules.

3. *Context Rules.* These rules return a new value for the *context* field, if this needs to be changed. These are run until either the set of context rules is exhausted or one of the rules returns a value other than the current context. Note that the priority is potentially important for determining which of several rules govern the new context.

4. *Trigger Rules.* These rules have a special predicate which sends a message to a process listening to the EIP. These rules are given both the old and new context values as often they will trigger when the context changes.

5. *Reset Rules.* These rules run only if the context changes. They are used to reset values of various timers and flags that should be reset for the new context.

As mentioned above, the context, verb, and object for an implicit part of the condition of the rule. These can take either a specific value, as defined by the application vocabulary, or the special keyword `All` indicating that this rule should be run no matter what the value of the corresponding field in the event or state. The context keyword can also be a context group defining a set of contexts to which this rule applies. Thus, for each event the EIP checks five collections of rules, each collection corresponding to the current context, verb and object as well as the type representing the phase of the event processing.

There is still a potential for sequence issues within each of these collections.[2] This is where the priority mechanism comes into play. The priority should be a positive integer. The rules are run in order of priority, lowest numbers going first. Ties are broken arbitrarily (and possibly in an implementation dependent way). By adjusting the priorities, the designer should be able to avoid potential conflicts and race conditions.

4.2 Conditions

The condition notation is adapted from the query documents used by the Mongo database (?, ?). The basic form is shown in Example **??**. The field should be either a field of the state or the event, using the notation described in Section **??**. The simplest form has the value as a simple value. The condition is satisfied if all of the fields have the indicated values, and not satisfied if one of them does not. Fields of the event or state which are not mentioned in the query document are ignored.

Example 4 Basic Condition Query Document

```
1       condition:{
2         <field1>:<value1>,
3         <field2>:<value2>,
4         ...
5       }
```

Instead of a simple value, the value part of the query document can be a more complex JSON object (enclosed in curly braces). The simplest version has the format: { *field1*:{ *?op*:*value1* }, ... }. The simple operators are: `?eq` (equality, usually omitted), `?ne` (not equals), `?gt` (greater than), `?gte` (greater than or equals), `?lt` (less than), and `?lte`. Note that these can be combined. For example `{ state.flag.objCount: { ?gt:5, ?lt:10}}` returns true when the `objCount` flag is between 5 and 10.

A slightly more complex version uses the form { *field1*:{ *?op*:[*value1a*, *value1b*,...] }, ... }. Here *?op* should be either `?in` or `?nin` (not in). These conditions are satisfied if the value of the field is (not) in the list. A shortcut is available for the `?in` operator: the expression { *field1*: [*value1a*, *value1b*,...], ... } is equivalent to more complicated version using `?in`.

Three special operators—`?exists`, `?isnull`, and `?isna`—test the state of the field. The first tests if the field exists, the second tests for a null value for the field and the third tests for a not-applicable or not-a-number value. In each case, the value should be either `true` or `false`. If true, the condition is satisfied if the field has the appropriate state, and if false, then if the field does not have that state.

[2] My recollection from trying to program in Prolog is that these sequencing issues were often the hardest part of a program to debug.

The `?any` and `?all` operators are used when the value being tested is an array. The argument for these operators should be a query with a single query. The `?any` test is satisfied if any of the elements of the array satisfy the test and the `?all` test is satisfied only if all of the elements of the array satisfy the test.

The `?not`, `?and`, and `?or` operators can be used to build more complex queries. The value for `?not` should be another query involving the target field; the not query is satisfied if the inner query is not satisfied. The `?and` and `?or` operators take an array (enclosed in `[]`) of queries about the current field. These are satisfied if all (or any) of the query documents in the array are satisfied. Note that unlike the Mongo database queries, the `?not`, `?and`, and `?or` operators apply only to a single field.

The `?regex` does regular expression matching: the argument should be a regular expression and the field should normally contain a string value. This is implementation dependent because different implementations will likely use the host regular expression engine, many of which have slight variations. (Simple regular expressions will likely work well in most implementations, but Perl or other extensions may or may not be supported.)

The final operator is `?where`, which actually appears as a field and not a value in the query document. The value should be a string giving the name of a function (with arguments `state` and `event`) which evaluates the state and event and returns true or false. This allows the handling of cases which cannot be easily handled by the query language (at the cost of being implementation dependent). It is strongly recommended that these query functions have not side effects (except for logging used when debugging).

4.3 Predicates

The predicate takes the generic form shown in Example **??**. The big difference is update operators, which control how the fields are modified. Fields should reference fields of the state object (dot notation beginning with `state`), although values can reference fields of the events.

Example 5 Basic Predicate Update Document

```
1       predicate:{
2         <update operator1>:{ <field1>:<value1>, ... },
3         <update operator2>:{ <field2>:<value2>, ... },
4         ...
5       }
```

The simplest update operator is the `!set` operator. In this case the field is set to the value (or if the value references an event or flag variable, to the value of that field). If the state object does not have a flag or observable of that name, it will be created.

The `!unset` operator is an inverse. For this the *value* should be one of `"NULL"`, `"NA"`, or `"Delete"`. The first two set the values to NULL and NA

(not applicable) respectively. The rules for these values are implementation dependent. The third removes the flag or observable from the state object.

The `!incr`, `!decr`, `!mult`, `!div`, `!min`, and `!max` operators modify the existing field (which should be numeric) based on the operator and the argument value. The first four add, subtract, multiply or divide the value of the field by the argument value and set the field to the result. The `!min` and `!max` operators set the field to the largest or smallest of the current value and the argument argument value.

The `!addToSet` and `!pullFromSet` operate on fields which are arrays. In the first case, the argument value is added to the set (if it is not already present), and in the second case it is removed from the set.

The `!push` and `!pop` operators also operate on array fields; however, they treat the arrays like stacks and not sets. The `!push` operator adds the argument value to the front of the list, and the `!pop` operator removes the first element, or if the value is a positive integer, it removes the that many elements from the front of the stack.

Timers are treated specially by the `!set` operator. In particular, there are two fields of the timer object which can be set, `state.timers.`*`name`*`.value` and `state.timers.`*`name`*`.running`. The latter value can be set to `true` or `false` which causes it to resume or pause respectively. The `.value` field of the timer sets the current time; this is assumed if the field is omitted. This can be treated like a numeric value with the time in seconds, or it can be set as an object of the form {`time:`*`number`*`, units:`*`unit`*}, where *unit* is one of `"sec"`, `"min"`, `"hr"`, `"day"` and so forth. Example **??** gives a few examples:

Example 6 Setting Timers

```
      predicate:{
        //Set pause learning support timer
        !set:{ state.timers.learnSup.running: false},
        //Resume level timer
        !set:{ state.timers.level.running:true},
        //Restart last play timer
        !set:{ state.timers.lastPlay.value: 0,
               state.timers.lastPlay.running: true},
        //Add 1 minute to the penalized time timer.
        !incr:{ state.timers.penTime : {time:1,
                                        units:"min"}},
        ...
      }
```

Finally, the `!setCall` operator provides an implementation dependent extension mechanism. The *value* should be a string giving the name of a function with three arguments: the name of the field being set, the state object and the event object. The named field will be set to the value returned by this function.

4.4 Special Predicates for Context and Trigger Rules

The context and trigger rules need special treatment. Context rules, if they fire return a string value naming the context. Consequently, the `predicate` component of Example **??** is replaced by a new `context` component. This should have a single value. It can be one of three things: (1) a string value for the new context, (2) the name of a field which provides the value for the new context, or (3) an expression of the form {`!call:`*`funname`*} which calls a program of the given name in the implementation language. This function should have two arguments, the state and the event, and return a string providing the name of the new context.

The role of the trigger rule is to send a message. The basic output message format looks like Example **??**. These are called `Proc4 messages` because the basic format came from an early implementation of the four process architecture (?, ?). When the message is sent, the `app` and `uid` values are set from the `state` object. The `timestamp` field is also set from the current time. The `context` field is often the value of `state.oldContext`, although that might be overridden.

Example 7 Proc4 Message Format

```
{
  app:<appid>,
  uid:<uid>,
  context:<contextVariable>,
  message:<message>,
  timestamp:<date-time>,
  data:{<field1>:<value1>, <field2>:<value2>,
      ...}
}
```

Consequently, the `predicate` component of a trigger rule is replaced with the `send` component, which specifies parts of the message. Example **??** provides the basic layout. The `message` field is required and should be a string. If the `context` field is omitted, it will default to the current value of `state.oldContext`. If the `data` field is omitted, it will default to all of the observables.

Example 8 Special Send Predicate for Trigger Rules

```
send:{
  message:<message>,
  listeners:<nameOrList>,
  context:<contextVariable>,
  data:{<field1>:<value1>, <field2>:<value2>,
      ...}
}
```

The `listeners` field requires more explanation. A number of other processes can register as listeners to messages send from the EIP. Normally, the EAP is a registered listener, but there might be other processes involved in logging, feedback generation and reporting, or event the ASP which want to listen to the messages. If this field is omitted, then the message is sent to all listeners. If a name of a specific listener or an array of listener names is supplied, then the message will be sent only two those listeners. This might be useful for error or debugging messages.

4.5 Rule Set Maintenance

Queries, adding rules to set. Removing rules from sets.
Defining context groups.

5 Examples

Please help me make some examples.

6 Software

The SVN repository for the current (very incomplete) implementation in R is available at `https://pluto.coe.fsu.edu/svn/common/EIEvent/`.