Rule of Evidence for Parsing Event Logs

Russell G. Almond
Florida State University

June 24, 2019

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

Almond, Steinberg, and Mislevy (2002) 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 (EI) Process Extracts evidence in the form of observed outcome variables from the raw work product.

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

Activity Selection (AS) Process 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 EI process.

Mislevy, Steinberg, and Almond (2003) defined an evidence model with two parts: the statistical model or *weights of evidence*—which correspond to the EA process,—and the *rules of evidence*—which corresponds to the EI process. Although the term *rules of evidence* is a pun on the legal term, literal rules of evidence were used in HyDRIVE (Mislevy & Gitomer, 1996), 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 2 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 3 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 1 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 (Betts & Smith, 2018). 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 EI process receives from the PP is an ordered sequence of these event records. (This could either be directly sent from the PP to the EI process or the EI process could retrieve them on demand from a learning record store.) The EI process processes these one at a time, updating the state of the system. In the end, the EI process must send one (or more) message to the EA process stating providing values for the observables seen in a particular context. This is what the EA process uses to update the student model.

2 Tasks and Contexts

Mislevy et al. (2003) 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.

Almond et al. (2002) 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 EI process. The EI process would process the work product and send the observable for that task to the EA process. The EA process would signal to the AS process 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.

Mislevy et al. (2015) (also Mislevy, 2013) 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

[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.

of a context depends on the assessment application. This section provides three examples of increasing complexity.

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 AS process selects a single item, which the PP displays, the EI process scores and the EA process updates the estimate of student ability. The AS process 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* (Shute & Ventura, 2013; Kim, Almond, & Shute, 2016) 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 EI process.

The *Physics Playground* PP logs events to a Learning Locker learning store ("Learning Locker Documentation", 2018). These events are recorded as JSON objects similar to the one shown in Example 1. 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 EI process 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 EI process 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 EA process (or other listeners). At this point, these are rules in the loose sense of Mislevy et al. (2003): requirements for the EI process. Section 4 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 EI process. 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 EI process 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 3.3 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 EI process. Section 3 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 EI process 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 EI process 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 EI process is useful even in contexts which are not simulations or games. In particular, Almond, Deane, Quinlan, Wagner, and Sydorenko (2012) 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 EI process can use to build up observables.

Although generally evidence rules are placed in the EI process, 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 EI process.

There may also be cases where the EA process 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 EA process, 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 EI process. In general, an observable is reported for one of three reasons:

1. It will be used by the EA process 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 (Attali & Burstein, 2006) 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 EI process 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.

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",
  "processed":false,
  "pError":null,
  "data":{
    "field1":"Value",
    "field2":["list","of","values"],
    "field3":{"part1":"complex","part2":"object"}
  }
}
```

3.3 Events

Look once more at the format of the event record in Example 1 (reproduced in Example 2). 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 EI process.

The `verb` and `object` fields should be keywords appropriate to the application. These fields are taken from the xAPI (Betts & Smith, 2018) data structure. The verb and object together should define what is happening. Some example pairs from *Physics Playground* include {(`"snapshot"`, `"ball"`), (`"exited"`, `"level"`), (`"exited"`, `"learning support"`), (`"identified"`, `"gameobject"`)}. 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 EI process 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.)

Note that two new fields have been added in Example 2: `processed` and `pError`. These allow a collection of events to be used as a queue. The can be

sorted by the `timestamp` and then as they are processed the `processed` field is set to true. The next event to process is the one with `processed=false` and the earliest timestamp. The field `pError` is set with an error message if an error occurs while processing the event.

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

Example 3 Generic Rule, JSON format.

```
{
  "name":"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":{...}
}
```

fields, which should improve the speed of execution. Example **??** shows a rule set in JSON format.

The condition (Section 4.2) and predicate (Section 4.3) are also JSON objects. They are adapted from the query language used by the Mongo database ("The MongoDB 4.0 Manual", 2018). The *type* and *priority* fields are used to determine the sequence in which rules are run. Sections 4.1 and 4.5 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. "Status" 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 EI process. 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.

Note that the `ruleType` field should be one of the five keywords "Status", "Observable", "Context", "Trigger" or "Reset".

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 EI process 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 MongoDB 4.0 Manual", 2018). The basic form is shown in Example 4. The field should be either a field of the state or the event, using the notation described in Section 3.4. 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.

`?eq`, `?ne`, `?gt`, `?gte`, `?lt`, `?lte` 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

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

Example 4 Basic Condition Query Document

```
"condition":{
  <field1>:<value1>,
  <field2>:<value2>,
  ...
}
```

`{ "state.flag.objCount": { "?gt":5, "?lt":10}}` returns true when the `objCount` flag is between 5 and 10.

`?in, ?nin` 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`.

`?exits, ?isnull, ?isna` 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.

`?any, ?all` 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.

`?not, ?and, ?or` 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.

`?regex` 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.)

`?where` 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).

Additional information about the query operators in the R implementation can be found by executing `help(Conditions)` at the command line.

4.3 Predicates

The predicate takes the generic form shown in Example 5. 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

```
"predicate":{
  <update operator1>:{ <field1>:<value1>, ... },
  <update operator2>:{ <field2>:<value2>, ... },
  ...
}
```

`!set` 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. (Timer objects, however, will not be created, see the `!start` operator.)

`!unset` 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.

`!incr, !decr, !mult, !div, !min, !max` 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.

`!addToSet` **and** `!pullFromSet` 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.

`!push, !pop` `!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. If the argument of `!pop` is a field reference, than that field is set to the value of on the top of the stack.

`!setKeyValue` This assumes that the target field is a named list (dictionary, or set of key–value pairs). It expects the argument to have fields `key:` and `value:` (which can be literal strings or references to fields) and will add a key–value pair to the target field. For example, the key could be the name of the level, and the value the trophy earned in the level. The final observable would be a list of all trophies earned, with the corresponding levels.]

`!set` **timer** 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 6 gives a few examples:

`!start, !reset` The `!set` operator assumes that timer object already has been created in the state object. The `!start` and `!reset` operations are synonyms for `!set` with some differences. First, both a `running` (or `run`) and `value` (or `time`) can be set at the same time. If only a real or POSIX time value is specified the it is assumed that the time should be set. If only a logical value is supplied, it is assumed that the running state should be set. If the logical value is not supplied, it is assumed to be `TRUE` for `!start` and `FALSE` for `!reset`. If the time value is not specified, it is assumed to be zero.

There is one important difference between the `!set` and the `!start` approach. They behave differently if the timer object is not already created in the state object. The `!set` operator (and related modification operators) will signal an error. The `!start` and `!reset` operators will create a new timer if needed.

`!setCall` Finally, the `!setCall` operator provides an implementation dependent extension mechanism. The *value* should be a string giving the name

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"}
                                        },
  ...
}
```

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.

Additional information about the predicate operators in the R implementation can be found by executing `help(Predicates)` at the command line.

4.4 Special Predicates for Context and Trigger Rules

Context rules are similar to status and observable rules. The difference is that if they fire, the predicate should set the `state.context` field. Example 7 gives the context rule used in *Physics Playground*.

The role of the trigger rule is to send a message. The basic output message format looks like Example 9. These are called `Proc4 messages` because the basic format came from an early implementation of the four process architecture (Almond et al., 2002). 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.

Consequently, the `predicate` component of a uses the special `!send` operator, which specifies parts of the message. (The `!send1` and `!send2` operators are used if multiple messages need to be sent with the same rule.) Example 9 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 7 Context Rule

```
{
  "name": "New Level Started",
  "doc": "Indicates that we should change level",
  "verb": "initialized",
  "object": "game level",
  "context": "ALL",
  "ruleType": "Context",
  "priority": 1,
  "conditions": {
    "state.context": {"?ne":"event.data.
       gameLevelId"}
  },
  "predicate": {
    "!set": {"state.context":"event.data.
       gameLevelId"}
  }
}
```

Example 8 Proc4 Message Format

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

Example 9 Special Send Predicate for Trigger Rules

```
"!send":{
  "mess":<message>,
  "context":<contextVariable>,
  "timestamp":<timestamp>,
  "data":{<field1>:<value1>, <field2>:<value2>,
     ...}
}
```

A number of other processes can register as listeners to the EI process (see the documentation for the `Proc4` process.) Normally, the EA process is a registered listener, but there might be other processes involved in logging, feedback generation and reporting. In *Physics Playground*, one listener passed messages on to the EI process and the other saved player state information used by the game engine. Generally, the listeners filter the messages they address based on the `mess` field, so this need to match what is expected.

4.5 Rule Set Testing and Maintenance

Configuration for the EI-Event engine consists of three collections of objects: a collection of rules, a collection of contexts (Section **??** and a default (initial) status. In the current implementation, these are stored in three collections in the database. In particular, the database indexes can be used to help rapidly find rules applicable for a context, or the particular context set.

In the current implementation, the `name` is regarded as a key, so they names should be unique throughout the collection. Loading two rules with the same name will replace the older rule. To prevent lots of warnings (and possible errors) the old rules are usually cleaned out before the replacements are loaded. As the rules are saved in the database, the loading can happen independently from the operational testing.

As the rules of evidence for a particular project is essentially a program, it needs to be tested and debugged. To that end, EI-Event defines a rule test object. Example 10 gives an example.

Example 10 Rule Test

```
1    {
2      "name": <Name used in reporting>,
3      "doc": <Human Readable Description>,
4      "initial": <Status object giving initial state>,
5      "event": <Event object to be tested>,
6      "rule": <Rule to be applied>,
7      "queryResult":<Logical value giving result of
           the query.",
8      "final": <Status object giving the final state>
9    }
```

These tests allow the rule authors to determine whether or not the rule is working as expected. Section 5 provides an example.

4.6 Context Tables

Recall that a rule is applicable to a given event if three conditions hold: (1) the `verb` field of the event and rule match, or the rule has the special verb "`ALL`", (2) the `object` field of the event and rule match, or the rule has the

special object "ALL", and (3) the context field of the status matches the rule, or the rule's context is a context set which contains the status's context (the context set "ALL" is a special set which contains all statuses). In order for this last matching to work properly, the context sets need to be defined. The context table is the mechanism for importing this information into the system.

Table 1 shows an excerpt from the file ContextSketching. There are three required columns: CID, Number and Name. For actual contexts (game levels), the Name and Number columns should be exactly as they are in the game engine. Any discrepancy in the name reported by the game and in the context table will result in an "unknown level" warning. Only ALL rules will be processed for that level. There are also some levels with negative numbers; these are context sets. The CID (context id) is a compressed version of the name which corresponds to variable naming conventions (e.g., no spaces or other special characters).

Table 1: Selected rows and columns from a context table.

CID	Number	Name	Sketching	RampLevels
INITIAL	0	*INITIAL*	0	0
Sketching	-100	Sketching Levels	0	0
RampLevels	-105	Ramp Levels	1	0
LeverLevels	-110	Lever Levels	1	0
CloudyDay	89	Cloudy Day	1	0
ClownChallenge	18	Clown Challenge	1	1
Cornfield	119	Cornfield	1	1
CosmicCave	112	Cosmic Cave	1	0
CrazySeesaw	95	Crazy Seesaw	1	0

Context sets should be given a row and assigned a negative number (the numbers should be unique). There should be a column in the spreadsheet corresponding to each of the context sets. The name of the column should be the CID of the corresponding level set. The contents of the column should be 0 or 1; 1 if the context in the row belongs to the set in the column and 0 otherwise. Additional columns can be placed in the spreadsheet for documentation purposes.

Finally, a special context "*INITIAL*" with number 0 is required. This is the context assigned the player when the system starts before the first event is received.

5 Examples

Consider the problem of counting how many times the player has manipulated the air resistance slider within a given level. The target observable is state.observables.airManip, the number of manipulations. At the beginning of the level, a reset rule is needed to set this observable to zero. Example 11 shows a rule which increments the counter.

Example 11 Count Air Resistance Manipulations Rule

```
{
  "app": "ecd://epls.coe.fsu.edu/PPTest"
  "name": "Count Air Resistance Manipulations",
  "doc": "Increment counter if slider changed.",
  "verb": "Manipulate",
  "object": "Slider",
  "context": "Manipulation Levels",
  "ruleType": "Observable",
  "priority": 5,
  "conditions": {
    "event.data.gameObjectType":"
       AirResistanceValueManipulator",
    "event.data.oldValue":{"?ne":event.data.newValue
       }
  },
  "predicate": {
    "!incr":{"state.observables.airManip":1}
  }
}
```

The `verb`, `object` and `context` determine when this rule is applicable. This rule is only checked if the verb of the event is "Manipulate" and the object is "Slider." Also, the context of the current state must be a game level that is in the "Manipulation Levels" set. (See Section 4.6). In particular, the rule is applicable to the event Example 12, if the current status has a context which is one of the "Manipulation Levels."

As the rule is applicable to this event and context, the conditions are checked. There are two conditions. First, the slider that was moved must be the Air Resistance slider (that is, `gameObjectType == "AirResistanceValueManipulator"`). Second, `oldValue != newValue`; that is, the player must have actually moved the slider.

As the conditions are met, the predicate is executed. Examples 13 and **??** show the before and after. In this case the `airManip` observable is incremented by one. Other flags, timers and observables remain the same. As this rule sets an observable, its type is "Observable". It is given a moderate priority of 5, as sequence is not particularly important.

Note that this example is in the form of a rule test (Section 4.5). However, additional tests are needed for this rule. In particular, it would need tests in which the conditions are not met and the rule does not increment the counter.

Example 12 Event for testing Air Manipulation Rule.

```
{
  "app":"ecd://epls.coe.fsu.edu/PPTest",
  "uid": "Test0",
  "verb": "Manipulate",
  "object": "Slider",
  "context": "Air Level 1",
  "timestamp":"2018-09-25 12:12:28 EDT",
  "data": {
    "gameObjectType": "
       AirResistanceValueManipulator",
    "oldValue": 0,
    "newValue": 5,
    "method": "input"
  }
}
```

Example 13 Initial State for testing the air manipulation rule.

```
{
  "app":"ecd://epls.coe.fsu.edu/PPTest",
  "uid": "Test0",
  "context": "Air Level 1",
  "timers":{},
  "flags":{
    "airUsed": true,
    "airVal": 5,
  },
  "observables":{
    "airManip":1
  }
}
```

Example 14 Final state after running Air Manipulation rule.

```
{
  "app":"ecd://epls.coe.fsu.edu/PPTest",
  "uid": "Test0",
  "context": "Air Level 1",
  "timers":{},
  "flags":{
    "airUsed": true,
    "airVal": 5
  },
  "observables":{
    "airManip":1
  }
}
```

6 Software

A reference implementation in R (R Core Team, 2018) is available at `https://pluto.coe.fsu.edu/Proc4/`.

While the R version is a complete version, it is also a first version, and there are a number of possible efficient improvements that can be made. However, the R implementation has a some serious limitations as R provides only limited support for parallel processing. (Being able to process different uids in different threads would speed up execution, but the R tools for multiprocessing are limited.) The next version is likely to be in a different language with better support for threading. On the other hand, the next version should still support the same rule JSON syntax (or at least provide an upward migration path).

References

Almond, R. G., Deane, P., Quinlan, T., Wagner, M., & Sydorenko, T. (2012). *A preliminary analysis of keystroke log data from a timed writing task* (Research Report No. RR-12-23). Educational Testing Service. Retrieved from `http://www.ets.org/research/policy_research_reports/publications/report/2012/jgdg`

Almond, R. G., Steinberg, L. S., & Mislevy, R. J. (2002). Enhancing the design and delivery of assessment systems: A four-process architecture. *Journal of Technology, Learning, and Assessment*, *1*, (online). Retrieved from `http://www.jtla.org/`

Attali, Y., & Burstein, J. (2006). Automated essay scoring with e-rater® v. 2.0. *The Journal of Technology, Learning, and Assessment*, *4*(3), 13–18. Retrieved from `http://escholorship.bc.edu/jtla/vol4/3/`

Betts, B., & Smith, R. (2018). The leraning technology manager's guid to xAPI (2nd ed.) [Computer software manual]. Retrieved from https://www.ht2labs.com/resources/the-learning-technology-managers-guide-to-the-xapi/#gf_26

Kim, Y. J., Almond, R. G., & Shute, V. J. (2016). Applying evidence-centered design for development of game-based assessments in Physics Playground. *International Journal of Testing*, *16*(2), 142-163. Retrieved from http://www.tandfonline.com/doi/ref/10.1080/15305058.2015.1108322 (Special issue on cogntive diagnostic modeling) doi: 15305058.2015.1108322

Learning locker documentation (2nd ed.) [Computer software manual]. (2018). Retrieved from https://docs.learninglocker.net/welcome/ (Retrieved 2018-09-03)

Mislevy, R. J. (2013). Evidence-centered design for simulation-based assessment. *Military Medicine*, *178*, 107-114.

Mislevy, R. J., & Gitomer, D. H. (1996). The role of probability based inference in an intelligent tutoring system. *User-Modeling and User-Adapted Interaction*, *5*, 253-282.

Mislevy, R. J., Oranje, A., Bauer, M. I., von Davier, A., Hao, J., Corrigan, S., ... Joh, M. (2015). *Psychometric considerations in game-based assessment* (Tech. Rep.). GlassLab: Institute of Play. Retrieved from http://www.instituteofplay.org/work/projects/glasslab-research/

Mislevy, R. J., Steinberg, L. S., & Almond, R. G. (2003). On the structure of educational assessment (with discussion). *Measurement: Interdisciplinary Research and Perspective*, *1*(1), 3-62.

The mongodb 4.0 manual (4.0 ed.) [Computer software manual]. (2018). Retrieved from https://docs.mongodb.com/manual/ (Retrieved 2018-09-03.)

R Core Team. (2018). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. Retrieved from https://www.R-project.org/

Shute, V. J., & Ventura, M. (2013). *Stealth assessment in digital games.* MIT series.