Package ‘EIEvent’

July 1, 2019

Version 0.4-2

Date 2019/07/01

Title Evidence Identification Event Processing Engine

Author Russell Almond

Maintainer Russell Almond <ralmond@fsu.edu>

Depends R (>= 3.0), methods, utils, R.utils, Proc4 (>= 0.4), jsonlite, mongolite, futile.logger

Description Extracts observables from a sequence of events.

License Artistic-2.0

URL https://pluto.coe.fsu.edu/Proc4

Collate AAGenerics.R Mongo.R Contexts.R Events.R Status.R Condition.R Predicates.R RuleTables.R testRules.R LL2Proc4.R EIEngine.R

R topics documented:

`EIEvent-package` *Evidence Identification Event Processing Engine*

Description

Extracts observables from a sequence of events.

Details

The DESCRIPTION file: This package was not yet installed at build time.

Index: This package was not yet installed at build time.

The package runs a `EIEngine` which is a server process for processing events according the the `EI-Event` rules stored in a database.

Configuration

The “quick start” document `https://pluto.coe.fsu.edu/Proc4/EIQuickStart.pdf` describes most of the configuration steps. The directory `file.path(help(library="EIEvent")$Path,"conf")` contains a number of files to assist with configuration.

The process assumes that Mongo (`https://www.mongodb.com/`) is installed on the system (or that it is available via a network). The files `setupMongo.js` and `setupUsers.js` (in the `conf` directory) are to be run in the mongo shell. In particular, the `setupMongo.js` sets up the indexes. Running these scripts is option, but recommended. See also the configuration instructions in the `Proc4` package.

It is also strongly recommended to set up a directory for configuration scripts for the packages in the Proc4 family. The recommended location on Unix systems is `/usr/local/share/Proc4`. The files `EIEvent`, `EIEvent.R`, `EIini.R`, and `EILoader.R` should be copied to that directory (or to a `bin` subdirectory), and edited for local prferences. In particular, the `EIini.R` file needs to be updated with local details about the database and passwords, as well as directories for configuration files. The other files need to be updated to point to the `EIini.R` file.

The `EILoader.R` script loads a scoring model into the engine (see the following section). The `EIEvent.R` script runs the scoring engine (see Launching and termination, below). The shell script `EIEvent` runs the `EIEvent.R` script as a server process.

Rule Sets and Context Sets

In many respects, the `EIEngine` is an interpreter for a rule-based programming langugage. The document *Rules of Evidence* (`https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`) describes that language specification. Programs are stored in the database. So there are two steps: loading the rules and contexts (described here), and executing the program on the event queue (described below).

The engine processes a queue of `Event` objects (see the next section) using the following steps:

1. When the first event for a user arrives, that user is defined an initial `Status` based on the default status for that application.
2. The `applicableContexts` for the current status is determined by consulting the `ContextSet` associated with the application. The idea is that a specific context (e.g., a game level) belongs to several larger context sets (e.g., the set of levels with a comon solution strategy).
3. The `RuleTable` is consulted to find the set of rules applicable to the current event and context.
4. The `Conditions` of the rules are run. If the condition is true, the the `Predicates` of the rules are run as well.
5. If running the rules updates the `Status`, then the updated status is saved.
6. If a trigger rule is run and `P4Message` is built (`!send`), then the message is sent to the `ListenerSet`.
7. The event is marked as processed, and then loop goes on to process the next event.

The program consists of three parts:

`RuleTable` A set of `Rule` objects which are all stored in the database.

`ContextSet` A set of context descriptions that play a role in determining when rules are applicable.

A defaul `Status` The initial status for the application.

The script `EILoader.R` illustrates the necessary steps.

Assuming the `EIEngine` is in an object called eng the following steps load the details.

1. Load in the rules from a JSON file: `ruleList <- lapply( fromJSON(`*filename* `, FALSE), parseRule)`. The function `parseRule` creates a rule object from the output of `fromJSON`.
2. Clear the old rules using `eng$clearAllRules()`.
3. Load the new rules using `eng$loadRules(ruleList)`.
4. Load the context set from a datafile: `conMat <- read.csv(`*filename*`)`.
5. Create the initial context: `initCon <- data.frame(CID="*INITIAL*", Name="*INITIAL*", Number=0)`.
6. Clear Old contexts: `eng$clearContexts()`.
7. Load context mattrix: `eng$addContexts(conMat)`.
8. Load initial context: `eng$addContexts(initCon)`.
9. Clear old records including default: `eng$clearStatusRecords(TRUE)`.
10. Create a new blank status: `defaultRec <- eng$newUser("*DEFAULT*")`.
11. Initialize fields on that record:
    - Flag: `flag(defaultRec,` *name* `) <-` *value*.
    - Observable: `obs(defaultRec,` *name* `) <-` *value*.
    - Timers: `timer(defaultRec,` *name* `) <- Timer(` *name*`)`.
12. Save it back to the database: `defaultRec <- eng$saveStatus(defaultRec)`.

Events and the Event Queue

Objects of class `Event` are stored in a database collection (by default, the `Events` collection in the `EIRecords` database). Each event has a `processed` flag and the `timestamp` ensure that at any time the system can find the oldest unprocessed event. When run in server mode, new events can be added to the collection and the `EIEngine` will process them as they are added.

A single cycle of the `mainLoop` consists of the following steps.

1. Fetch the next event with `eve <- eng$fetchNextEvent()`.
2. Process the event with `out <- handleEvent(eng,eve)`.
3. If an error was generated (out is of class `try-error`), then save the error in the database using `eng$setError(eve, out)`.
4. Mark the event as processed using `eng$setProcessed(eve)`.

Events can be added to the system using the `mongoimport` external function. To call this from R, use `system2("mongoimport", sprintf('-d %s -c Events --jsonArray', eng$dbname), stdin=eventFil`

Listeners and Messages

The output of the EI process is through the `Listener` objects. Trigger rules generate objects of `P4Message` (see `!send`) which are sent to the `ListenerSet` associated with the `EIEngine`. When a trigger rule fires, the `!send` predicate creates a message which is then sent to the ListenerSet though the `notifyListeners` method. This, in turn, calls the `receiveMessage` method on each of the listeners.

One of the most common actions in response to a listener firing is to insert or update a record in another database. This is the mechanism by which the EI process can send its output into the input queue for the EA process.

Launching and termination

Because the program is already stored in the database, launching the `EIEngine` as a server process takes three steps:

1. Create an instance of `EIEngine` (eng).
2. Create set the `active` flag for the process to true by calling, `eng$activate()`.
3. Run the `mainLoop` process. This will run until the `active` flag is cleared.

The script `EIEvent.R` runs through these steps. It calls the `EIini.R` script to get details of configuration, then creates the engine and runs the main loop. The bash shell script `EIEvent` runs the `EIEvent.R` script in a server process (`R --slave`).

When run from the command line, `EIEvent` takes four arguments. (These should be specified as `name=value`). They are as follows:

`app` **(Required)** This is the name of the application to be run. It is usually a URL-like string such as: `ecd://epls.coe.fsu.edu/P4test`.

`level` **(Default `INFO`)** This is the default logging level (see Logging below).

`clean` **(default `FALSE`)** If true, then old statuses and messages will be cleaned out before starting. Otherwise, the current run will be a continuation of the old run.

`evidence` **(Optional, filename)** If this argument is supplied, then the contents of the file will be read into the event queue before processing. The value of `eng$processN` will be set to the number of events, so the function will stop after the events are processed.

The arguments are only relevant when the process is run as a server. The `EIEvent.R` script can also be run line-by-line in an R developement environment (e.g., R Studio), in which case the arguments are replaced by constants set at the beginning of the script.

Unless an event file is given, the script will run an infinite loop. On a *nix system, this usually means that it should run as a background process, e.g., the call should be `nohup EIEvent args &`. This requires a graceful way to close the process down. This is done by setting the `active` field for the record corresponding to the application to false in the `AuthorizedApps` collection of the `Proc4` database. A mongo shell call to do this is: `db.AuthorizedApps.update({app:{"$regex":"appname"}}, {"$set":{ac` where "appname" can be any string that uniquely identifies the application.

If the event file is given, it should be a JSON file will a collection of `Event` objects. The events will be loaded into the database and processed. The script will stop when the events are done processing. Often it is more convenient to run this in the interactive mode as R will still be active after the completion, so that the results can be inspected.

Logging and Error Handling

Logging is handled using the `flog.logger` framework. This provides a large number of tools for specifying the details of the logging. The `EIEngine` executes the rules in the `withFlogging` environment. This means that the default behavior for rules which generate errors is to log the error and move to the next rule. The error message is also added to the event in the event database (`markAsError`).

The amount of logging done, particularly the amount of detail supplied, is controled by the `flog.threshold` function. The amount of detail provided at various levels is as follows:

TRACE
- The results of `checkCondition` are reported.
- The specific rules found from each query are reported.
- A message is logged as each rule is run.

DEBUG
- When an error occurs information about the state, event and rule where the error occurred as well a stack trace are logged.
- Each event is logged as it is processed.
- Rule searches are logged and the number of rules reported.
- A message is logged when each phase starts.
- A message is logged when the context changes.

INFO Minimal information about which events are being processed is logged.

WARN and above If an error occurs the error is logged along with context information.

When running the `EIEngine` as a server, generally the logging should be done as a file. This can be done by running: `flog.appender(appender.file(logfile))`. When running in interactive mode, it may be useful to have log messages sent to both the console and the log file. The command for this is: `flog.appender(appender.file(logfile))`.

Concurrency

Note that several applications can share the same database. The `app` field of the `EIEngine` is part of the key for all of the collections. Generally, database commands will use the app field as part of the queries, so two engines with different applications ids will have different rule sets, context sets, status sets and event queues.

Because of this separation, it is possible to run multiple `EIEngine` processes focused on different applications. This gives a limited form of parallel processing.

In theory, processing for each `uid` could be handled separately. This has not been coded in this version. Future versions may use a language other than R which supports richer tools for multi-threaded computing.

Acknowledgements

Work on the Proc4, EIEvent and EABN packages has been supported by the National Science foundation grants *DIP: Game-based Assessment and Support of STEM-related Competencies* (#1628937, Val Shute, PI) and *Mathematical Learning via Architectual Design and Modeling Using E-Rebuild.* (#1720533, Fengfeng Ke, PI).

The EIEvent package developement was led by Russell Almond (Co-PI).

Author(s)

Russell Almond

Maintainer: Russell Almond <ralmond@fsu.edu>

References

The document “Rules Of Evidence” gives extensive documentation for the rule system. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

There is a “quick start” document which describes how to set up the engine. This is available at https://pluto.coe.fsu.edu/Proc4/EIQuickStart.pdf.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

See Also

The package `Proc4` provides low level support for the database connectivity, which is handled through the `mongo` function. Logging is supplied by the `flog.logger` system.

Examples

```
## Not run:

## Initialize the Engine
app <- "ecd://epls.coe.fsu.edu/P4test"
loglevel <- "DEBUG"
cleanFirst <- TRUE
eventFile <- "/home/ralmond/Projects/EvidenceID/c081c3.core.json"

## These configuration steps are generally done in EIini.R
## These are application generic parameters
EIeng.common <- list(host="localhost",username="EI",password="secret",
                     dbname="EIRecords",P4dbname="Proc4",waittime=.25)

appstem <- basename(app)
## These are for application specific parameters
EIeng.params <- list(app=app)

## File for loading configuraitons
config.dir <- "/home/ralmond/ownCloud/Projects/NSFCyberlearning/EvidenceID"

## Location of logfile
logfile <- file.path("/usr/local/share/Proc4/logs",
                     paste("EI_",appstem,"0.log",sep=""))

EI.listenerSpecs <-
  list("InjectionListener"=list(sender=paste("EI",appstem,sep="_"),
            dbname="EARecords",dburi="mongodb://localhost",
            colname="EvidenceSets",messSet="New Observables"))
```

```
## Setup logging
flog.appender(appender.file(logfile))
flog.threshold(loglevel)

## Setup Listeners
listeners <- lapply(names(EI.listenerSpecs),
                    function (ll) do.call(ll,EI.listenerSpecs[[ll]]))
names(listeners) <- names(EI.listenerSpecs)
if (interactive()) {
  cl <- new("CaptureListener")
  listeners <- c(listeners,cl=cl)
}

## Load the Rules
ruleList <- lapply(fromJSON("rulefile.json", FALSE), parseRule)
eng$clearAllRules()
eng$loadRules(ruleList)

## Load Contexts
conMat <- read.csv("contextTable.csv")
initCon <- data.frame(CID="*INITIAL*", Name="*INITIAL*", Number=0)
eng$clearContexts()
eng$addContexts(conMat)
eng$addContexts(initCon)

## Setup default status.
eng$clearStatusRecords(TRUE) ## Clears default record
defaultRec <- eng$newUser("*DEFAULT*")
obs(defaultRec,"bankBalance") <- 0
defaultRec <- eng$saveStatus(defaultRec)

## Clean out old records from the database.
if (cleanFirst) {
  eng$eventdb()$remove(buildJQuery(app=app(eng)))
  eng$userRecords$clearAll(FALSE)   #Don't clear default
  eng$listenerSet$messdb()$remove(buildJQuery(app=app(eng)))
  for (lis in eng$listenerSet$listeners) {
    if (is(lis,"UpdateListener") || is(lis,"InjectionListener"))
      lis$messdb()$remove(buildJQuery(app=app(eng)))
  }
}
## Process Event file if supplied
if (!is.null(eventFile)) {
  system2("mongoimport",
          sprintf('-d %s -c Events --jsonArray', eng$dbname),
          stdin=eventFile)
  ## Count the number of unprocessed events
  NN <- eng$eventdb()$count(buildJQuery(app=app(eng),processed=FALSE))
  ## This can be set to a different number to process only a subset of events.
  eng$processN <- NN
}

## Activate engine (if not already activated.)
```

```
eng$activate()
mainLoop(eng) ## This will not terminate unless processN was set to a
              ## finite value.

## This shows the details of the last message.  If the test script is
## set up properly, this should be the observables.
if (!is.null(eventFile) && TRUE) {
  details(cl$lastMessage())
}


## End(Not run)
```

`applicableContexts` *Finds context sets to which a given context belongs.*

Description

A `Context` may belong to one or more context sets. A `Rule` may operate on a specific context, a context set, or all contexts (the special context set `"ALL"`). The function `applicableContexts` returns a list of all potential rule context matches. The function `belongsTo` maintains the implicit contexts.

Usage

```
applicableContexts(c)
belongsTo(c)
## S4 method for signature 'Context'
belongsTo(c)
## S4 method for signature 'ANY'
belongsTo(c)
belongsTo(c) <- value
## S4 replacement method for signature 'Context'
belongsTo(c) <- value
```

Arguments

`c`	An object of class `Context`
`value`	A character vector containing the names of the context sets this context belongs to.

Value

The function `belongsTo` returns a (possibly empty) vector of context set names this context belongs to. (The `ANY` method always returns the empty list.) The function `applicableContexts` returns the same vector with the addition of the current context ID and the special context `"ALL"`.

Note

It would seem a natural extension of this system to put the contexts into an acyclic directed graph with the `belongsTo` function providing the link. This was determined to be more trouble than it was worth for the current application, so the entire hierarchy must be represented within the `belongsTo` field of each context.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the context system: https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

See Also

`Context` describes the context object.

Examples

```
ct <- Context("Level1","First Tutorial",1,
              belongsTo=c("tutorialLevels","easyLevels"),
              doc="First Introductory Level",
              app="ecd://epls.coe.fsu.edu/EITest")
stopifnot(setequal(belongsTo(ct),c("tutorialLevels","easyLevels")))
stopifnot(setequal(applicableContexts(ct),
          c("Level1","tutorialLevels","easyLevels","ALL")))

belongsTo(ct) <- "tutorialLevels"
stopifnot(setequal(belongsTo(ct),c("tutorialLevels")))
```

`asif.difftime` *More flexible constructor for creating difftime objects.*

Description

The function `asif.difftime` is a constructor for `difftime` objects from a list with components named “secs”, “mins”, “hours”, “days”, or “weeks”. These get added together.

The function `is.difftime` is the test function missing from the base package.

Usage

```
asif.difftime(e2)
is.difftime(x)
```

Arguments

`e2`	This should be a list of numeric values with named components with names selected from: `c("secs", "mins",   "hours", "days", "weeks")`.
`x`	An object to be tested for its difftime status.

Value

If the argument to `asif.difftime` is a list with the appropriate names, then an object of class difftime is returned. Otherwise, the argument is returned.

The function `is.difftime` returns a logical value indicating whether or not its argument is of class `difftime`

Author(s)

Russell Almond

See Also

`difftime`4

Also, `asif.difftime` is used in variuos predicates.

Examples

```
dt <- asif.difftime(list(mins=1,secs=5))
stopifnot (is.difftime(dt),
    all.equal(dt,as.difftime(65,units="secs")))
```

`buildMessages` *These functions build messages for Trigger Rules.*

Description

A trigger rule has a special predicate function `!send` which build a `P4Message` object to send to listeners of the `EIEngine` (through the `notifyListeners` method.

Usage

```
buildMessages(predicate, state, event)
"!send"(predicate, state, event)
"!send1"(predicate, state, event)
"!send2"(predicate, state, event)
```

Arguments

`predicate`	One element of the predicate, or in the case of `buildMessages` the complete predicate.
`state`	An object of class `Status` giving the current state of the system. Note that often `context` and `oldContext` will be different.
`event`	An object of class `Event` that gives the event that triggered the message.

Details

The function `!send` builds a `P4Message` which is then sent to the registered listeners of the `EIEngine`. The default message is “"Observables Available"”, which is what the Evidence Accumulation process is listening for. The default content is all of the observables.

The rule can override certian defaults of the message by setting appropriate fields of the object that is the value of the `!send` element in the predicate.

context The context for the message. Default is `oldContext(event)`. Value can be a direct value of a field name (starting with “event.data.” or “state.”).

mess The text (subject) of the message. Default is “Observables Available”. Value can be a direct value of a field reference (starting with “event.data.” or “state.”).

data The fields to be added in the details section of the message. If omitted, all of the observables are sent (with their default names). If supplied this should either be an named character vector or named list. The names are used as the names of the details portion of the message, and the values are field references (starting with “event.data.” or “state.”) for where to find the values.

The function `buildMessages` runs through multiple elements of the predicate and builds multiple messages. The additional predicates `!send1` and !send2 are there to allow for additional messages. This will also work with the name of an arbitrary R function which takes `(predicate, state, event)` as an argument list and returns a message.

Value

The function `!send` returns an object of class `P4Message` with the following fields:

`app`	The application, value is `app(event)`.
`uid`	The user ID, value is `uid(event)`.
`context`	The context (task) for this message, default is the value of `oldContext(state)`.
`mess`	The message header. The default value is “Observables Available.”
`sender`	The value is “EIEvent”.
`timestamp`	The timestamp for the message. The value is `timestamp(event)`.
`details`	The data that should be sent with the message. The default is all of the observables of the `Status` argument.

Note

JSON parsers are not happy with multiple fields with the same name, so the `!send1` and `!send2` operators are provided to get around this problem.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: `https://education.umd.edu/file/11333/download?token=kmOIVIwi`, Video: `https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4`.

See Also

`Rule` describes the rule object and Conditions describes the Conditions, and Predicates the predicates in general.

The functions `testRule` and `testRuleScript` can be used to test that rule conditions and predicates function properly together.

Other classes in the EIEvent system: `EIEngine`, `Context`, `Status`, `Event`, `RuleTable`.

The `P4Message` class is from the Proc4-package. The `notifyListeners` method describes the Proc4 message passing system.

Examples

```
st9 <- Status(uid="Test0",context="SpiderWeb",
             timestamp=as.POSIXct("2018-09-25 12:13:30 EDT"),
             observables=list(agentsUsed=list("Pendulum"),
                              lastAgent="Pendulum",
                              badge="gold"))


evnt10 <- Event(uid="Test0",
               verb="satisfied",object="game level",
               context="SpiderWeb",
               timestamp=as.POSIXct("2018-09-25 12:13:30 EDT"),
               details= list("badge"="gold"))

mess0 <- buildMessages(list("!send"=list()),st9,evnt10)[[1]]
stopifnot(mess(mess0)=="Observables Available",
          length(details(mess0)) == 3L)
mess1 <- buildMessages(list("!send1"=list()),st9,evnt10)[[1]]
messb <- buildMessages(list("!send"=list(mess="Badge",
                 data=c("badge"="state.observables.badge"))),
         st9,evnt10)[[1]]
stopifnot(mess(messb)=="Badge",length(details(messb))==1L)
```

`checkCondition` *Checks to see if a condition in a EIEvent Rule is true.*

Description

An `Rule` object contains a list of Conditions. The name of each condition is the name of a field of the `Event` or `Status` object. For example, `"event.data.trophy"=list("?in"=c("gold","silver"))` would test if the trophy field was set to “gold” or “silver”. The function `checkCondition` returns true if all of the conditions are satisfied and false if any one of them is not satisfied.

Usage

```
checkCondition(conditions, state, event)
```

Arguments

`conditions`	A named list of conditions: see details.
`state`	An object of class `Status` to be checked.
`event`	An object of class `Event` to be checked.

Details

The condition of a `Rule` is a list of queries. Each query has the following form:

field`=list(`*?op*`=`*arg*`,...)`

Here, *field* is an identifier of a field in the `Status` or `Event` object being tested. This is in the dot notation (see `getJS`). The query operator, *?op* is one of the tests described in Conditions. The *arg* is a value or field reference that the field will be tested against. In other words, the query is effectively of the form *field ?op arg*. The `...` represents additional `?op`–`arg` pairs to be tested.

The *arg* can be a literal value (either scalar or vector) or a reference to another field in the `Status` or `Event` object using the dot notation.

In general, a rule contains a list of queries. A rule is satisfied only if all of its queries are satisfied: the function `checkCondition` checks if the rule is satisfied.

Finally, one special query syntax allows for expansion. If the *field* is replaced with `"?where"`, that is the query has the syntax `"?where"=`*funname*, then the named R is called. This should be a function of two arguments, the status and the event, which returns a logical value. The condition is satisfied if the funciton returns true.

See Conditions for more details.

Value

Returns a logical value: `TRUE` if all conditions are satisfied, false otherwise.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. https://docs.mongodb.com/manual/.

See Also

`Rule` describes the rule object and Conditions describes the conditions. Predicates describes the predicate part of the rule, and `executePredicate` executes the predicate (when the condition is satisfied).

The functions `testQuery` and `testQueryScript` can be used to test that rule conditions function properly.

Other classes in the EIEvent system: `EIEngine`, `Context`, `Status`, `Event`, `RuleTable`.

Examples

```
st <- Status("Phred","Level 1",timerNames=character(),
   flags=list(lastagent="lever",noobj=7,noagents=0),
   observables=list(),
   timestamp=as.POSIXct("2018-12-21 00:01"))


ev <- Event("Phred","test","message",
      timestamp=as.POSIXct("2018-12-21 00:01:01"),
      details=list(agent="lever",newobj=2))


stopifnot(
checkCondition(list(event.data.agent=list("?eq"="lever")),
               st,ev)==TRUE,            #Agent was lever.
checkCondition(list(event.data.agent="ramp"),
               st,ev)==FALSE,           #Agent abbreviated form.
checkCondition(list(event.data.agent="state.flags.lastagent"),
               st,ev)==TRUE,            #Same agent used.
checkCondition(list(state.flags.noobj=list("?lt"=10,"?gte"=5)),
               st,ev)==TRUE,            #Between 5 and 10 objects.
checkCondition(list(event.data.agent=list("?in"=c("pendulum","springboard"))),
               st,ev)==FALSE,           #Between 5 and 10 objects.
#Abbreviated form (note lack of names)
checkCondition(list(event.data.agent=c("lever","springboard")),
               st,ev)==TRUE,
checkCondition(list(state.flags.lastagent=list("?isna"=TRUE)),
               st,ev)==FALSE,
checkCondition(list(state.flags.noagents=
                    list("?and"=list("?isna"=FALSE,"?gt"=0))),
               st,ev)==FALSE
)

agplusobj <- function (state,event) {
  return (getJS("state.flags.noobj",state,event) +
          getJS("event.data.newobj",state,event) < 10)
}

stopifnot(checkCondition(list("?where"="agplusobj"),st,ev))
```

`cid` *Accessor functions for context objects.*

Description

These functions access the corresponding fields of the `Context` class.

Usage

```
cid(c)
## S4 method for signature 'Context'
cid(c)
number(c)
## S4 method for signature 'Context'
number(c)
number(c) <- value
## S4 replacement method for signature 'Context'
number(c) <- value
## S4 method for signature 'Context'
app(x)
```

Arguments

c	A `Context` object.
x	A `Context` object.
value	An integer giving the new context number.

Value

The function `cid` returns a unique string identifier for the context. The function `number` returns a unique integer identifier. The function `app` returns the application identifier. The `cid` and `number` should be unique within the app.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the JSON layout of the Status/State objects. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: `https://education.umd.edu/file/11333/download?token=kmOIVIwi`, Video: `https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4`.

Betts, B, and Smith, R. (2018). The Leraning Technology Manager's Guid to xAPI, Second Edition. HT2Labs Research Report: `https://www.ht2labs.com/resources/the-learning-technology-managers-guide-to-` `#gf_26`.

HT2Labs (2018). Learning Locker Documentation. `https://docs.learninglocker.net/welcome/`.

See Also

`Context` describes the context object. The function `applicableContexts` describes the context matching logic.

Examples

```
ct <- Context("Level1","First Tutorial",1,
              belongsTo=c("tutorialLevels","easyLevels"),
              doc="First Introductory Level",
              app="ecd://epls.coe.fsu.edu/EITest")

stopifnot(cid(ct)=="Level1",basename(app(ct))=="EITest",
          number(ct)==1L)

number(ct) <- 0
stopifnot(number(ct)==0L)
```

`Conditions`	*Conditional query operators for Rules.*

Description

These are the conditional operators for a `Rule`. The check that the `target` value meets the specified condition (`cond`). These are called by the function `checkCondition` which checks that the condition is correct.

Usage

```
"?eq"(arg, target, state, event)
"?ne"(arg, target, state, event)
"?gt"(arg, target, state, event)
"?gte"(arg, target, state, event)
"?lt"(arg, target, state, event)
"?lte"(arg, target, state, event)
"?in"(arg, target, state, event)
"?nin"(arg, target, state, event)
"?exists"(arg, target, state, event)
"?isnull"(arg, target, state, event)
"?isna"(arg, target, state, event)
"?regexp"(arg, target, state, event)
"?any"(arg, target, state, event)
"?all"(arg, target, state, event)
"?not"(arg, target, state, event)
"?and"(arg, target, state, event)
"?or"(arg, target, state, event)
```

Arguments

`arg`	This is the value of the condition clause (the argument) of the query.
`target`	This is the value of the current state of the referenced field in the query.
`state`	This is a `Status` object used to resovle dot notation references.
`event`	This is a `Event` object used to resovle dot notation references.

Details

The condition of a `Rule` is a list of queries. Each query has the following form:

field`=list(`*?op*`=`*arg*`,...)`

Here, *field* is an identifier of a field in the `Status` or `Event` object being tested. This is in the dot notation (see `getJS`). The query operator, *?op* is one of the tests described in the section ‘Condition Operators’. The *arg* is a value or field reference that the field will be tested against. In other words, the query is effectively of the form *field ?op arg*. The `...` represents additional ?op–arg pairs to be tested.

The *arg* can be a literal value (either scalar or vector) or a reference to another field in the `Status` or `Event` object using the dot notation.

In general, a rule contains a list of queries. A rule is satisfied only if all of its queries are satisfied (essentially joining the queries with a logical-and). At the present time, the only way to get a logical-or is to use multiple rules.

Finally, one special query syntax allows for expansion. If the *field* is replaced with `"?where"`, that is the query has the syntax `"?where"`=*funname*, then the named R is called. This should be a function of two arguments, the status and the event, which returns a logical value. The condition is satisfied if the funciton returns true.

Value

The condition operators always return a logical value, `TRUE` if the query is satisfied, and `FALSE` if not.

Condition Operators

The syntax for the condition part of the rule resembles the query lanugage used in the Mongo database (MongoDB, 2018). There are two minor differences: first the syntax uses R lists and vectors rather than JSON objects and second the '$' operators are replaced with '?' operators.

In general, each element of the list should have the form *field*=`c(`*?op*=*arg*`)`. In this expression, *field* references a field of either the `Status` or `EIEngine` (see sQuoteDot Notation section above), *?op* is one of the test operators below, and the argument *arg* is a litteral value (which could be a list) or a character string in dot notation referencing a field of either the `Status` or `Event`. If *?op* is omitted, it is taken as equals if *arg* is a scalar and *?in* if value is a vector. For more complex queries where `arg` is a more complex expresison, the `c()` function is replaced with `list()`.

The following operators (inspired from the operators used in the Mongo database, Mongo DB, 2018, only with '?' instead of '$') are currently supported:

- `?eq`, `?ne` These are the basic operators, which test if the field is (not) equal to the argument.
- `?gt`, `?gte`, `?lt`, `?lte` These test if the field is greater than (or equal to) or less than (or equal to) the argument. Note that `c("?lt"=low,"?gt"=high)` can be used to test if the value of the field is between the arguments *low* and *high*.
- `?in`, `?nin` These assume that the argument is a vector and are satisfied if the value of the field is (not) in the vector.
- `?exists`, `?isnull`, `?isna` These test if the field exists, contains a `NULL` (often true if the field does not exist) or contains an `NA`. The *arg* should be `TRUE` or `FALSE` (where false inverts the test.)
- `?any`, `?all` These operators assume that the field contains a vector and check if any (or all) of the elements satisfy the condition given in the argument. Thus, the argument is another expression of the form `=c(`*?op*=*value*`)`. For example *field*=`list("?any"=c("?gt"=3))` will be satisfied if any element of *field* is greater than 3.
- `?not` The argument of this query should be another query involving the target field. The not query is satisfied if the inner query is not satistified.
- `?or`, `?and` In both of these cases, the *arg* should be a list of queries for the applicable field. The `?or` query is satisfied if any of the inner queries is satisfied, and the `?and` query is satistified if all of the inner queries are satisfied. Like the R `||` (`&&`) operator, the `?or` (`?and`) query runs the subqueries in order and stops at the first true (false) subquery.

`?regex` This query uses regular expression matching. The argument should be a regular expression (see regex for a description of R regular expressions). The query is satisfied if the value of the field matches the regular expression.

`?where` This query is a trapdoor that allows arbitrary R code to be run to run as the condition. This has a special syntax: `"?where"`=*funname*, where the `?where` operator takes the place of the field and *funname* give the name of a function. This should be a function of two arguments, the status and the event, which returns a logical value. The condition is satisfied if the funciton returns true.

Although the `?or` operator allows for logical-or expressions for a single field, it does not extend to multiple fields. However, this can be accomplished with separate rules.

Expansion Mechanisms

The special “?where” form obviously allows for expansion. It is particularly designed for queries which involve multiple fields in a complex fashion.

It is also possible to expand the set of *?op* functions. The method used for dispatch is to call `do.call(op,list(cond, target, state, event))` where *cond* is everything after *?op*=*cond* in the query expression. (This is the same syntax as the supplied operators).

Condition Testing

The function `checkCondition` is used internally to check when a set of conditions in a rule are satisfied.

The functions `testQuery` and `testQueryScript` can be used to test that rule conditions function properly. The functions `testRule` and `testRuleScript` can be used to test that rule conditions and predicates function properly together.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: `https://education.umd.edu/file/11333/download?token=kmOIVIwi`, Video: `https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4`.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. `https://docs.mongodb.com/manual/`.

See Also

Rule describes the rule object and Predicates describes the predicates. The function checkCondition tests when conditions are satisfied. The functions testQuery and testQueryScript can be used to test that rule conditions function properly.

Other classes in the EIEvent system: EIEngine, Context, Status, Event, RuleTable.

Examples

```
list(
event.data.agent=list("?eq"="lever"),   #Agent was lever.
event.data.agent="lever",               #Agent abbreviated form.
event.data.agent=list("?ne"="ramp"),   #Agent was not a ramp.
event.data.agent="state.flag.lastagent", #Same agent used.
state.flags.noobj=list("?lt"=10),       #Fewer than 10 objects used.
state.flags.noobj=list("?lt"=10,"?gte"=5), #Between 5 and 10 objects.

#Agent is a lever or a springboard.
event.data.agent=list("?in"=c("lever","springboard")),
#Abbreviated form (note lack of names)
event.data.agent=c("lever","springboard"),
#Agent was not a ramp or a pendulum.
event.data.agent=list("?nin"=c("ramp","pendulum")),

## Checking for existence of fields and NA values.
state.timers.learningsupport=list("?exists"=TRUE),
event.data.newvalue=("?isnull"=FALSE),
state.flags.lastagent=list("?isna"=TRUE),

## Was the slider a blower (name starts with blower).
event.data.slider=list("?regexp"="^[Bb]lower.*"),

## These assume field is a vector.
state.flags.agentsused=list("?any"=list("?eq"="pendulum")),
state.flags.agentsused=list("?any"="pendulum"), #Abbreviated form.
state.flags.agentsused=list("?all"=list("?eq"="ramp")),
state.flags.agentsused=list("?any"="ramp"), #Abbreviated form.

## ?not
state.flags.agentsused=list("?not"=list("?any"="ramp")),

## ?and, ?or -- note these stop as soon as falsehood (truth) is proved.
state.flags.noagents=list("?and"=list("?is.na"=FALSE,"?gt"=0)),
state.flags.noagents=list("?or"=list("?is.na"=TRUE,"?eq"=0))

)

## The ?Where operator
agplusobj <- function (state,event) {
  return (getJS("state.flags.noobj",state,event) +
          getJS("event.data.newobj",state,event) < 10)
}

list("?where"="agplusobj")
```

Context *Constructor for the Context object*

Description

This is the constructor for the Context objects and context set objects (which are identical). As Context objects are usually read from a database or other input stream, the parseContext function is recreates an event from a JSON list and as.jlist encodes them into a list to be saved as JSON.

Usage

```
Context(cid, name, number, belongsTo = character(), doc = "", app="default")
parseContext(rec)
## S4 method for signature 'Context,list'
as.jlist(obj, ml, serialize = TRUE)
```

Arguments

cid	A character identifier for the context. Should be unique within an application (app).
name	A human readable name, used in documentation.
number	A numeric identifier for the context.
belongsTo	A character vector describing context sets this context belongs to.
doc	A character vector providing a description of the context.
app	A character scalar providing a unique identifier for the application.
rec	A named list containing JSON data.
obj	An object of class Context to be encoded.
ml	A list of fields of obj. Usually, this is created by using attributes(obj).
serialize	A logical flag. If true, serializeJSON is used to protect the data field (and other objects which might contain complex R code).

Details

Most of the details about the Context object, and how it works is documented under Context-class. Note that context sets and contexts are represented with the same object.

The function as.jlist converts the obj into a named list. It is usually called from the function as.json.

The parseContext function is the inverse of as.jlist applied to a context object. It is designed to be given as an argument to getOneRec and getManyRecs.

Value

The functions `Context` and `parseContext` return objects of class `Context`. The function `as.jlist` produces a named list suitable for passing to `toJSON`.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the context system: `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: `https://education.umd.edu/file/11333/download?token=kmOIVIwi`, Video: `https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4`.

Betts, B, and Smith, R. (2018). The Leraning Technology Manager’s Guid to xAPI, Second Edition. HT2Labs Research Report: `https://www.ht2labs.com/resources/the-learning-technology-managers-guide-to-`
`#gf_26`.

HT2Labs (2018). Learning Locker Documentation. `https://docs.learninglocker.net/welcome/`.

See Also

`Context` describes the context object, and `ContextSet` describes the context set object.

`parseMessage` and `as.json` describe the JSON conversion system.

The functions `getOneRec` and `getManyRecs` use `parseStatus` to extract events from a database.

Examples

```
ct <- Context("Level1","First Tutorial",1,
              belongsTo=c("tutorialLevels","easyLevels"),
              doc="First Introductory Level",
              app="ecd://epls.coe.fsu.edu/EITest")

cta <- parseContext(as.jlist(ct,attributes(ct)))
stopifnot(all.equal(ct,cta))
```

Context-class *Class* "Context"

Description

This is a descriptor for a measurement context (e.g., item or game level) in an assessment system. A context plays the role of a *task* in the four process architecture (Almond, Steinberg, and Mislevy, 2002), but allows for the measurement context to be determined dynamically from an extended task. (Almond, Shute, Tingir, and Rahimi, 2018).

The primary of use of the `Context` object is determining for which events a rule is applicable. The `belongsTo` allows designers of an evidence rule system to define context groups for rules which are applicable in multiple contexts.

Objects from the Class

Context object can be created by calls to the `Context()`) function. Most of the fields in the context are documentation; however, two, the `cid` and `belongsTo` fields, play a special role in the rule dispatch logic. The `cid` field is the identifier of the context as used in the `Rule` and `Status` classes.

The `belongsTo` attribute sets up a hierarchy of classes. In particular, each value of the of this field (if set) should be the `cid` of a context group to which this context belongs. Context groups are also `Context` objects and can in turn belong to larger groups. Currently, inheritance is not supported, so that the all parents (direct and indirect) need to be listed).

Context Resolution

When the `EIEngine` processes an `Event`, it checks the context of the current `Status` object. It then searches the rule table for all rules which match on the `verb` and `object` fields of the event and the `context` field of the status. A rule is considered applicable if one of the following conditions is met.

1. The `context` fields of the `Status` and `Rule` class match exactly.
2. The `context` fields of the `Rule` class matches the `cid` of one of the entries in the `belongsTo` field of the `Status`.
3. The `context` fields of the `Rule` is the keyword “ANY”, *i.e.*, the rule is applicable to all classes.

This is actually accomplished by using the function `applicableContexts` which creates a list of the current context and all of the context groups that it matches. The `EIEngine` then grabs from the rule table all rules which match one of the applicable contexts (including “ANY”).

This should seem similar to the way that method dispatch works for S4 classes (see Introduction to the methods package). The `Context` `belongsTo` hiearchy is similar to the inheritence hierarchy of a typical class system. There are two key differences. First, inheritense is not currently supported with contexts; all context groups must be explicitly listed in the `belongsTo` field. Second, while the S4 method dispatch mechanism searches for all methods which are applicable to the current objects, it only executes the most specific method. The table dispatch mechanism executes *all* the applicable rules and makes no attempt to sort them.

Slots

- `_id`**:** Object of class `"character"` which is the id in the Mongo database; this generally should not be changed.
- `cid`**:** Object of class `"character"` which provides a unique identifier for the context.
- `name`**:** Object of class `"character"` which provides a human readable name for the context.
- `number`**:** Object of class `"integer"` which provides a numeric index for the context.
- `belongsTo`**:** Object of class `"character"` which gives a list of context IDs for context groups to which this context belongs.
- `doc`**:** Object of class `"character"` which provides extended documentation for the context group.
- `app`**:** Object of class `"character"` which a unique identifier for the application in which this context is applicable.

Methods

- **as.jlist** `signature(obj = "Context", ml = "list")`: Used in converting the object to JSON for storing in a Mongo database, see `as.json`.
- **belongsTo** `signature(c = "Context")`: Returns the value of the `belongsTo` field
- **belongsTo<-** `signature(c = "Context")`: Sets the value of the `belongsTo` field.
- **cid** `signature(c = "Context")`: Returns the context ID.
- **doc** `signature(x = "Context")`: Returns the documentation string.
- **name** `signature(x = "Context")`: Returns the name of the context.
- **number** `signature(c = "Context")`: Returns the number id of the context.
- **number<-** `signature(c = "Context")`: Sets the numeric id of the context.
- **app** `signature(x = "Context")`: Returns the app identifier of the context.
- **show** `signature(object = "Context")`: Prints the object in a "#<Context >" format.
- **toString** `signature(x = "Context",...)`: Converts the object to a string in a "#<Context >" format.

Note

It seems natural to create a full inheritance hierarchy for contexts. Probably available in a future version. For now, explicity listing all parents seems easier to implement.

Author(s)

Russell Almond

References

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: `https://education.umd.edu/file/11333/download?token=kmOIVIwi`, Video: `https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4`.

See Also

The `ContextSet` is an object which hold a collection of contexts.

Other classes in the EIEvent system: `EIEngine`, `Event`, `Status`, `Rule`.

Methods for working with contexts: `Context`, `applicableContexts`, `parseContext`, `name`, `doc`, `cid`, `number`, `app`

Examples

```
showClass("Context")
```

`ContextSet-class` *Class* `"ContextSet"`

Description

This is a reference class which is a wrapper for a handle to a database collection of `Context` objects associated with a particular application. Note that this is a reference class and many of the methods permanently modify the database.

Extends

All reference classes extend and inherit methods from `"envRefClass"`.

Methods

clearContexts `signature(set = "ContextSet")`: Removes all of the contexts associated with this applicaiton from the database.

matchContext `signature(id = "character", set = "ContextSet")`: searches for a `Context` with the given context id (`cid`).

matchContext `signature(id = "integer", set = "ContextSet")`: searches for a `Context` with a given numeric id (`number`).

updateContext `signature(con = "Context", set = "ContextSet")`: Adds or replaces a `Context` in the database.

ContextSet `signature(app = "character", dbname = "character", dburi = "character", ...)`: Constructor, creates a new context set for the given application, connecting to the database referenced by `dbname` and `dburi`.

Fields

`app`**:** Object of class `character` giving the identifier for the application. This is part of the database key for the collection.

`dbname`**:** Object of class `character` giving the name of the database (e.g., “EIRecords”).

`dburi`**:** Object of class `character` giving the uri for the database, (e.g., “mongodb://localhost”).

`db`**:** Object of class `MongoDB` which is the handle to the database. As this field is initialized when first requested, it should not be accessed directly, but instead through the `contextdb()` method.

Class-Based Methods

`update(con)`: This inserts of replaces a `Context` object in the database.

`initialize(app, dbname, dburi, db, ...)`: This sets up the object. Note that the db field is not initialized until `contextdb()` is first called.

`contextdb()`: This returns the handle to the `mongo` collection object. If the connection has not yet been initialized.

`named(id)`: This searches for the context by context id (`cid`), not by name.

`numbered(num)`: This searchers for the context by number.

`clearAll()`: This removes all contexts associated with this application from the database.

Note

In general, several context sets can share the same database collection. These are distinguished by the application ID (`app`). The effect primary key for the collection is `(app,cid)` with a secondary key as `(app,number)`. The collection maintains an index on both of those field pairs.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the context system: `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

See Also

`Context` describes the context object.

The functions `matchContext`, `updateContext` `clearContexts`, and `loadContexts` are used to manipulate context sets.

Examples

```
showClass("ContextSet")
```

`doc` *Meta-data accessors for Rules and Contexts.*

Description

To provide both `Rule` and `Context` object with adequate documentation, they are given both `name` and `doc` properties. The `name` is used in error reporting and debugging. The `doc` string is to aid in rule management.

Usage

```
doc(x)
## S4 method for signature 'Context'
doc(x)
## S4 method for signature 'Rule'
doc(x)
name(x)
## S4 method for signature 'Context'
name(x)
## S4 method for signature 'Rule'
name(x)
```

Arguments

x	A `Context` or `Rule` object whose documentation is being queried.

Value

Both functions return an object of type character.

Author(s)

Russell Almond

See Also

Classes in the EIEvent system: `EIEngine`, `Context`, `Status`, `Event`, `Rule`.

Examples

```
r1 <- Rule(name="Coin Rule",
           doc="Set the value of the badge to the coin the player earned.",
           app="ecd://coe.fsu.edu/PPtest",
           verb="satisfied", object="game level",
           context="ALL",
           ruleType="Observable", priority=5,
           condition=list("event.data.badge"=c("silver","gold")),
           predicate=list("!set"=c("state.observables.badge"=
                                   "event.data.badge")))

stopifnot(name(r1)=="Coin Rule",grepl("badge",doc(r1)))

ct <- Context("Level1","First Tutorial",1,
              belongsTo=c("tutorialLevels","easyLevels"),
              doc="First Introductory Level",
              app="ecd://epls.coe.fsu.edu/EITest")
stopifnot(name(ct)=="First Tutorial",
          grepl("Intro",doc(ct)))
```

EIEngine — *Creator for the EIEngine class.*

Description

The EIEngine is the prime mover for the EIEvent class. This command constructs the engine. The engine stores most of its information in a database, so the constructor mainly consists of the database location and credentials.

Usage

```
EIEngine(app = "default", listeners = list(), username = "", password = "", host = "localhost", port = ""
```

Arguments

`app`	The application ID for the engine. See Details.
`listeners`	A list of objects of type `Listener` which will receive messages from the engine.
`username`	A username for the database. If left blank the database will be accessed with an anonymous user.
`password`	A password for the database. If left blank, the database will be accessed without a password.
`host`	The hostname for the database. Defaults to “localhost”.
`port`	This is the port used for the database connection. If left blank, the default port 27017 will be used.
`dbname`	This the name of the database. The default name is “EIRecords”.
`P4dbname`	Database used for checking if the EIEngine should still be active.
`processN`	A positive integer. When the `mainLoop` is started, the engine will process N records before stopping.
`...`	For future expansions, subclasses.

Details

The `EIEngine` is an interpreter for the rules, which form a code base for a specific engine. The `EIEngine` class provides connection to six collections in a database which provide most of the action of the EIEvent system.

Events Incomming events (`Event` objects). Accessed through `$eventdb()` method.

Messages Outgoing messages (`P4Message` objects). Accessed through `ListenerSet` (`$listenerSet`) object.

Rules Rule collection (`Rule` objects). Accessed through `RuleTable` (`$rules`) object.

States Collection of saved statuses (`Status` objects). Accessed through `UserRecordSet` (`$userRecords`) object.

Contexts Context collection (`Context` objects). Accessed through `ContextSet` (`$contexts`) object.

Tests Self-test collection (`EITest` objects). Accessed through `TestSet` (`$tests`) object.

Note that the database connections are made in a lazy fashion, connecting to the database when first accessed rather than when the object is created. So the proper accessor methods (e.g., `$eventdb()` whould be used instead of the raw field names.

All of the collections have the `app` field as part of their key, so that several applications can share a database. A different EIEngine is needed for each application.

Value

A reference object of class `EIEngine`. See the class page for information about the methods supported.

Note

This class is not currently threadsafe. In particular, if the `handleEvent()` function is called simultaneously on two different events with the same `uid`, the result will not be pretty. However, separate applications can be run in parallel (in different processes or threads).

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: `https://education.umd.edu/file/11333/download?token=kmOIVIwi`, Video: `https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4`.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. `https://docs.mongodb.com/manual/`.

See Also

The primary classes in the EIEvent system are: `EIEngine`, `Context`, `Status`, `Event`, `Rule`.

The EIEngine class is a container for the following classes: `UserRecordSet`, `RuleTable`, `ContextSet`, `TestSet` and `ListenerSet`,

`flog.logger`

`mainLoop`, `handleEvent`

Examples

```
cl <- new("CaptureListener")
eng <- EIEngine(app="ecd://epls.coe.fsu.edu/EItest",listeners=list(capture=cl))
```

`EIEngine-class` *Class* `"EIEngine"`

Description

This is the main worker class for the EIEvent process. It handles grapping messages and processing them.

Extends

All reference classes extend and inherit methods from `"envRefClass"`.

Methods

notifyListeners `signature(sender = "EIEngine")`: This sends a message to the registered listeners of the EIEngine.

app `signature(sender = "EIEngine")`: Returns a string giving the application this Engine is supporting.

Database Connections

The `EIEngine` is an interprer for a rule-based langauge for processing events. The code, the rules, are stored in a `mongo` database, as are the events (the input queue) and the status objects. This database is determined by the `dburi` (note that this is determined with the `username`, `password`, `host` and `port` of the `$initialize()` method or `EIEngine` function) and `dbname`. Note that this database can be shared by several applications. The `app` field is used to identify records belonging to this application.

In the default configuration, the database is called "EIRecords" and has the following collections:

Events This contains the `Event`s to be processed. It can be accessed through the function `$eventdb()`. Note that each event has a `processed` field that makes this collection essentially a queue. Other processes can queue events to be processed by inserting event records into this collection.

***Messages*:** This is used by the `ListenerSet` to record messages sent by trigger rules. This collection can be accessed throught the `$listenerSet` field.

***Rules*:** This contains the collection of `Rule`s. This collection can be accessed through the `$ruleTable` field.

***States*:** This contains the collection of `Status` objects. This collection can be accessed through the `$userRecords` field.

***Context*:** This contains the collection of `Context` objects. This can be accessed through the `$contexts` field.

***Tests*:** This contains the collection of `RuleTest` objects. This can be accessed through the `$ruleTests` field.

The `AuthorizedApps` collection in the `Proc4` database is used to signal when the EI process should shut down. The initialization argument `P4dbname` is used to set the name of this database, and the `P4db()` function to access the collection. Note that all database collections should be accessed through the accessor function, which inialize the connection on demand, rather than by accessing the field directly.

The listener objects may also access databases. This is the usual mechanism for the EI process to insert messages into the processing queue of other processes.

Activation and Termination

The `mainLoop` searches the event database for unprocessed events, and processes them in chronological order. There are two termination conditions. First, if the `$processN` field (which is decrimented after each event is processed) reaches 0, then the main loop will exit. Second, if the event queue is empty, the engine will check the `active` flag in the record for its process in the `AuthorizedApps` collection. This is done with a call to `$isActivated()` which returns the value of the flag.

The method `$activate()` sets the activated flag for this app to true. This should be called before starting the `mainLoop`.

The flag must be manually set to false. This generally cannot be done from R, as R is busy running the main loop. The following command run in the mongo shell will cause the main loop to gracefully exit: `db.AuthorizedApps.update({app:{"$regex":"P4test"}},   {"$set":{active:false}});`, where “P4test” should be replaced with any string uniquely identifying the app.

Events and the Event Queue

Objects of class `Event` are stored in a database collection `$eventdb()`. The `processed` flag and the `timestamp` ensure that at any time the system can find the oldest unprocessed event. The function `$fetchNextEvent()` fetches this event. When the event is processed, the function `$setProcessed()` can be used to mark it as processed an update the database. The function `$setError()` is used to add an error message to an event.

A single cycle of the `mainLoop` consists of the following steps.

1. Fetch the next event with `eve <- eng$fetchNextEvent()`.
2. Process the event with `out <- handleEvent(eng,eve)`.
3. If an error was generated (out is of class `try-error`), then save the error in the database using `eng$setError(eve,        out)`.
4. Mark the event as processed using `eng$setProcessed(eve)`.

Note that `$setProcessed()` still needs to be called even if `$setError()` is called.

Rules

Each application has a collection of rules or a `RuleTable`. This rule table is stored in the `$rules` field of the `EIEngine` object. Recall that a rule is applicable to a given event if the `verb` and `object` of the event match the verb and object of rule (or the rule has the special object "ALL") and the `context` of the rule is one of the `applicableContexts` of the current context of the state.

In the current implementation, the rules are stored in the `Rules` database. This solves two problems: rule storage and rule search.

First, the database is where the rules come from when the application starts. The method `$loadRules()` stores a list of rules in the database and `$clearAllRules()` removes all rules. Note that trying to add a second rule with the same name will produce an error, unless the second argument (`stopOnDups`) is set to `FALSE`. So generally rules need to be cleared before being reloaded. (For finer replacement see the `RuleTable` object.) Typically, loading the rules is done in three steps:

1. Load in the rules from a JSON file: `ruleList <- lapply( fromJSON(`*filename*`, FALSE), parseRule)`. The function `parseRule` creates a rule object from the output of `fromJSON`.
2. Clear the old rules using `eng$clearAllRules()`.
3. Load the new rules using `eng$loadRules(ruleList)`.

Second, the search for applicable rules can use standard database queries. The function `$findRules()` searches the database for rules matching the supplied details. Note that the `EIEngine` version of this function differs from the `RuleTable` version in that it accepts a string as input to for the `context` field and finds a context object to match, while the `RuleTable` version expects an object of class `Context`. The `phase` argument is optional. The first version of the main loop, used the database to separate out rules of different types. However, this was rather inefficient when there were not applicable methods for a rule (the most common case). The current version now searches for rules that would be applicable in any phase of processing and then separates the resulting rule lists.

The `$findRules()` and rule processing functions do a fair amount of logging, so they can be debugged by setting `flog.threshold` to a lower value. At the `DEBUG` level, the number of rules returned for each event is logged. At the `TRACE` level, the queries used as well as the names of the returned rules are logged.

Context Sets

Rules must be applicable to both the current event and in the current `Status`. The `verb` and `object` fields of the rule can either be a constant, which is matched as an exact string, or the special constant "ALL" which indicates that the rule is applicable to all verbs or objects.

For `Context`s, a second intermediate level is introduced: the *context set*. The context object defines which context sets it belongs to through the `belongsTo` field. Context objects are stored in the database and are matched to the `context` field of events using the `name` field. (The context field of events is trimmed on input to prevent issues with mismatches caused by trailing spaces, see `parseMessage`.)

The field `$contexts` is a link to a `ContextSet` class which contains all of the contexts. The function `$getContext()` which takes the name of the context as an argument, calls `matchContext` to find the context object, and returns the context object. Note that if the string is not matched, it will behave like a context which only belongs to the universal context set "ALL".

Contexts are loaded by parsing a context matrix, a `data.frame` with the following columns: `cid` (character, required), `number` (numeric, required), `name` (character, optional) and `doc` (character, required). Additional columns should represent context sets, and should be 1 if the context row is in the set column and 0 otherwise. See `loadContexts` for an example. A special context with number 0 and cid "*INITIAL*" is required to represent the initial state when the player first logs into the system. The method `$addContexts()` adds the contexts in the matrix to the system. The method `$clearContexts()` removes existing contexts.

A typical initialization has the following steps:

1. Load the context set from a datafile: `conMat <- read.csv(`*filename*`)`.
2. Create the initial context: `initCon <- data.frame(CID="*INITIAL*", Name="*INITIAL*", Number=0)`.
3. Clear Old contexts: `eng$clearContexts()`.
4. Load context mattrix: `eng$addContexts(conMat)`.
5. Load initial context: `eng$addContexts(initCon)`.

Status and Default Status

The `$userReocrds` field of the engine points to a `UserRecordSet` collection. This is a set of `Status` records for the students. Once again, it is stored in the database to give the status persistence across invocations of the engine.

The method `$getStatus()` fetches a status from the database for the specified user. If no status is found, then the method `$newUser()` is called to create the inital status. This will look for a status for a user called "*DEFAULT*", and will copy that status if available. Otherwise, a blank status (one with the `timers`, `flags` and `obsservables` fields empty.

The method `$saveStatus()` saves the status back to the database. Note that the return value should be captured, as this will update the `_id` (see `m_id`) field of the record. The method `$clearStatusRecords(clearDefault)` will remove old status records, so that all players start fresh. With the argument set to true, it will also remove the default record.

The following steps can be used to set up the default record.

1. Clear old records including default: `eng$clearStatusRecords(TRUE)`.
2. Create a new blank status: `defaultRec <- eng$newUser("*DEFAULT*")`.
3. Initialize fields on that record:
   - Flag: `flag(defaultRec,` *name* `) <-` *value*.
   - Observable: `obs(defaultRec,` *name* `) <-` *value*.
   - Timers: `timer(defaultRec,` *name* `) <- Timer(` *name*`)`.
4. Save it back to the database: `defaultRec <- eng$saveStatus(defaultRec)`.

Listeners and Messages

The output of the EI process is through the `Listener` objects. The `$listenerSet` field holds an object of class `ListenerSet` which in turn holds the listeners. When a trigger rule fires, the `!send` predicate creates a message which is then sent to the ListenerSet though the `notifyListeners` method. This, in turn, calls the `receiveMessage` method on each of the listeners.

During construction, the initialization method of the `EIEngine` is passed a named list of listener objects. (It is important that all of the listeners be named, as the ListenerSet iterates over them by name.)

The ListenerSet object maintains a connection to the `Messages` collection in the database used by the `EIEngine`. All messages are logged to this database. The listeners, in contrast, generally only record messages whose message matches a given criteria. Many of the listeners connect to a database other than the internal one: They could either insert messages in the queue for another process or update a record in a database that tracks player status.

The `CaptureListener` is particularly useful for testing. It stores all messages in a list and has a specific method for viewing the latest one.

Main Event Loop

The goal of the `EIEngine` is to act as a server process. It continually scans the event queue, looking for unprocessed events. It processes the oldest unprocessed event, marks it as processed and then looks for the next event. If no unprocessed events are found it sleeps for a bit and then tries again. The field `$waittime` is the amount of time in seconds that the main loop will wait before checking again.

The function `mainLoop` implements the main processing loop. Generally, it should be the last call in a script for running the process (see `EIEvent.R` in the `config` subdirectory of the package for an example). It terminates under one of two conditions: (1) When the event queue is empty, it checks the `$isActivated()` method which checks the flag in the `AuthorizedApps` collection in the `Proc4` database. If the function returns false, the main loop will terminate. (2) The `$processN` field is decremented with every event processed. If it is given a finite value (the default is infinity) then it will process that many records and then stop.

Running `mainLoop` with `$processN` set to a positive integer is useful for testing. In one mode a collection of events can be read into the database from a test file, then the `$processN` field can be set to the number of new events (see `EIEvent.R` for an example). This will then run the test events and then stop. Alternatively, `$processN` can be set to a number lower than the total number of events and then the logging threshold can be set higher, or the processing can be run step by step to isolate problematic events.

Logging and Error Handling

Logging is handled using the `flog.logger` framework. This provides a large number of tools for specifying the details of the logging. The `EIEngine` executes the rules in the `withFlogging` environment. This means that the default behavior for rules which generate errors is to log the error and move to the next rule. The error message is also added to the event in the event database (`markAsError`).

The amount of logging done, particularly the amount of detail supplied, is controled by the `flog.threshold` function. The amount of detail provided at various levels is as follows:

TRACE
- The results of `checkCondition` are reported.
- The specific rules found from each query are reported.
- A message is logged as each rule is run.

DEBUG
- When an error occurs information about the state, event and rule where the error occurred as well a stack trace are logged.

- Each event is logged as it is processed.
- Rule searches are logged and the number of rules reported.
- A message is logged when each phase starts.
- A message is logged when the context changes.

INFO Minimal information about which events are being processed is logged.

WARN and above If an error occurs the error is logged along with context information.

When running the `EIEngine` as a server, generally the logging should be done as a file. This can be done by running: `flog.appender(appender.file(logfile))`. When running in interactive mode, it may be useful to have log messages sent to both the console and the log file. The command for this is: `flog.appender(appender.file(logfile))`.

Test Sets

This is mostly a placeholder for future capability. Along with the rules, which are after all a program, there should be a test suite. The goal is to provide a collection of tests, so that the test suite can be rerun when rules are modified, creating a unit testing facility for the *EI-Event* language.

This has not yet been implemented, but the function `testRuleScript` provides a mechanism for writing test scripts.

Tracing

The EIEngine uses the `flog.logger` system. In particular, setting the threshold for the "EIEvent" logger, will change the amount of information sent to the log file.

Fields

`app`**:** Object of class `character`. This is the long url-like name of the application. Note that it is a key to all of the database lookup.s

`dburi`**:** Object of class `character` giving the URI for the link to the Mongo database

`dbname`**:** Object of class `character` giving the name of the user database

`P4dbname`**:** Name of the database which contains the `AuthorizedApps` collection used to start and stop the engine.

`p4db`**:** Object of class `MongoDB` referencing the `AuthorizedApps` collection used to start and stop the engine. This field should not be referenced directly, instead, the `P4db()` should be used as it will initialize the connection if needed.

`waittime`**:** The number of seconds to wait before rechecking the queue when it is empty.

`userRecords`**:** Object of class `UserRecordSet` giving the user record set for the applcation.

`rules`**:** Object of class `RuleTable` giving the rule set for the application

`contexts`**:** Object of class `ContextSet` giving the set of contexts supported by the application.

`events`**:** Object of class `MongoDB` a mongo DB connection which allows access to events. Note that this is created when first requested, so should be accessed using the eventsdb() method.

`ruleTests`**:** Object of class `TestSet` give a collection of tests for the rule system.

`listenerSet`**:** Object of class `ListenerSet` giving the registered listeners for messages sent by the system.

`processN`**:** An integer counting how many events the `mainLoop` should process before stopping. If infinite, the main loop will not stop until the authorized app flag is cleared.

Class-Based Methods

`initialize(app, listeners, username, password, host, port, dbname, waittime, processN, ...)`: This method sets up the object. Note that the listeners object is passed in as an argument, as are the database credentials.

`P4db()`: Returns a handle to the `AuthorizedApps` collection in the Proc4 database. Initializes if needed.

`isActivated()`: Returns true if the active flag is set on this applicaiton (`app`) in the `AuthorizedApps` collection.

`activate()`: Sets the active flag to true for this `app` in the `AuthorizedApps` collection.

`show()`: Provides a short string identifying the engine.

`newUser(uid)`: Generates a new student record. (Call to `UserRecordSet$newStudent()`)

`getStatus(uid)`: Feteches the current status for a student from the database. (Call to `UserRecordSet$getStatus()`)

`saveStatus(state)`: Saves an updated status to the database. (Call to `UserRecordSet$saveStatus()`)

`clearStatusRecords(clearDefault)`: Clears the status records, including the default record if the argument is TRUE. (Call to `UserRecordSet$clearAll()`).

`getContext(id)`: This is a call to `matchContext`.

`addContexts(conmat)`: This is a call to `loadContexts`.

`clearContexts()`: This is a call to `ContextSet$clearAll()`.

`eventdb()`: This returns the event database handle, creating it if it is not yet created. It is recommended to use this method rather than access the slot directly.

`setProcessed(mess)`: This sets the `processed` flag on its argument and updates the database.

`setError(mess,e)`: This sets the `processingError` flag on `mess` to `e` and updates the database..

`fetchNextEvent()`: This returns the unprocessed event with the oldest timestamp, or NULL if there are no unprocessed events.

`findRules(verb, object, context, phase=NULL)`: This function finds the potentially applicable for the current verb, object, context and phase. If `phase==NULL`, the get rules for all phases.

`loadRules(rlist, stopOnDups)`: This function loads rules into the rule table using `loadRulesFromList`.

`loadAndTest(script, stopOnDups)`: This function loads rules into the rule table from test scripts using `testAndLoad`.

Note

A different EIEninge process is associated with each application (application ID), so different applicaitons can run in parallel. More sophisticated concurancy checking is not currently available.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the rule system. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

There is a “quick start” document which describes how to set up the engine. This is available at https://pluto.coe.fsu.edu/Proc4/EIQuickStart.pdf.

See Also

The function `EIEngine` tells about the construction and the relationship between the Engine and the database.

The primary classes in the EIEvent system are: `EIEngine`, `Context`, `Status`, `Event`, `Rule`.

The EIEngine class is a container for the following classes: `UserRecordSet`, `RuleTable`, `ContextSet`, `TestSet` and `ListenerSet`,

`flog.logger`

Examples

```
showClass("EIEngine")
```

`EITest-class` *Class* `"EITest"`

Description

An object describing a test case for a certain evidence identification system. It describes the input state, the event and the target output state so that the interaction of the rules can be checked.

Objects from the Class

An EITest object consists of a test. The test case has an *initial* `Status`, a triggering `Event`, and an expected *result* `Status`. If the goal is intead to check output messages, the result can be a `P4Message` or a list of such messages.

Objects can be created by calls of the form `EITest(...)`, or from JSON through `parseEITest`.

Slots

`_id`**:** Object of class `"character"`, the internal mongo identifier.

`app`**:** Object of class `"character"`, the unique identifier for the application to which this belongs.

`name`**:** Object of class `"character"`, a human readable name for the test, used in reporting.

`doc`**:** Object of class `"character"`, a documentation string describing the test.

`initial`**:** Object of class `Status` which describes the initial state of the system before the test.

`event`**:** Object of class `Event` which describes the incoming event to which the system is reacting.

`final`**:** Object of class `Status` which describes the expected final state of the system after applying the rules, or an object of class `P4Message` describing the generated message.

Methods

as.jlist `signature(obj = "EITest", ml = "list")`: helper function for converting the test into a JSON object, see `as.json`.

doc `signature(x = "EITest")`: Returns the documenation string.

event `signature(x = "EITest")`: Returns the triggering event for the test.

final `signature(x = "EITest")`: Returns the expected final state of the system.

initial `signature(x = "EITest")`: Returns the initial state of the test.

name `signature(x = "EITest")`: Returns the name of the test.

show `signature(object = "EITest")`: Displays the rule object.

toString `signature(x = "EITest")`: Returns a string describing the rule.

Author(s)

Russell G. Almond

References

The document “Rules Of Evidence” gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

See Also

The functions `EITest` and `parseEITest` are used to construct the rule.

The functions `name`, `doc`, `initial`, `event`, and `final` access the components of the test.

The `TestSet` maintains a collection of tests for a particular application.

Examples

```
showClass("EITest")
```

`Event` *Event object constructor*

Description

The `Event` funciton is the constructor for the `Event` object. As Event objects are usually read from a database or other input stream, the `parseEvent` function is recreates an event from a JSON list.

Usage

```
Event(uid, verb, object = "", timestamp = Sys.time(), details = list(), app = "default", context = "", pr
parseEvent(rec)
## S4 method for signature 'Event,list'
as.jlist(obj, ml, serialize = TRUE)
```

Arguments

`uid`	A character scalar identifying the examinee or player.
`verb`	A character scalar identifying the action which triggered the event.
`object`	A character scalar identifying the direct object of the verb.
`timestamp`	An object of class `POSIXt` which provides the time at which the event occurred.
`details`	A named list of detailed data about the the event. The available fields will depend on the `app`, `verb` and `object`.
`app`	A character scalar providing a unique identifier for the application (game or assessment). This defines the available vocabulary for `verb` and `object`, as well as the set of applicable `Rule` objects.
`context`	A character string describing the task, item or game level during which the event occurred. It could be blank if the context needs to be figured out from surrounding events.
`processed`	A logical flag. Set to true after the event has been processed by the `EIEngine`.
`rec`	A named list containing JSON data.
`obj`	An object of class `Event` to be encoded.
`ml`	A list of fields of `obj`. Usually, this is created by using `attributes(obj)`.
`serialize`	A logical flag. If true, `serializeJSON` is used to protect the `data` field (and other objects which might contain complex R code.

Details

Most of the details about the `Event` object, and how it works is documented under Event-class.

The function `as.jlist` converts the `obj` into a named list. It is usually called from the function `as.json`.

The `parseEvent` function is the inverse of `as.jlist` applied to an event object. It is designed to be given as an argument to `getOneRec` and `getManyRecs`.

Value

The functions `Event` and `parseEvent` return objects of class event. The function `as.jlist` produces a named list suitable for passing to `toJSON`.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the JSON layout of the Event objects. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

Betts, B, and Smith, R. (2018). The Leraning Technology Manager's Guid to xAPI, Second Edition. HT2Labs Research Report: https://www.ht2labs.com/resources/the-learning-technology-managers-guide-to-
#gf_26.

HT2Labs (2018). Learning Locker Documentation. https://docs.learninglocker.net/welcome/.

See Also

Event describes the event object.

parseMessage and as.json describe the JSON conversion system. In particular, as Event extends P4Message, the Event method for `as.jlist` calls the P4Message method.

The functions getOneRec and getManyRecs use `parseEvent` to extract events from a database.

Examples

```
ev1 <- Event("Phred","test","message",
      timestamp=as.POSIXct("2018-12-21 00:01:01"),
      details=list("list"=list("one"=1,"two"=1:2),"vector"=(1:3)))
ev2 <- Event("Phred","wash","window",
      timestamp=as.POSIXct("2018-12-21 00:02:01"),
      details=list(condition="streaky"))
ev3 <- Event("Fred","open","can",
      timestamp=as.POSIXct("2018-12-21 00:03:01"),
      details=list(lidOn=FALSE))


ev1a <- parseEvent(ununboxer(as.jlist(ev1,attributes(ev1))))
ev2a <- parseEvent(ununboxer(as.jlist(ev2,attributes(ev2))))
ev3a <- parseEvent(ununboxer(as.jlist(ev3,attributes(ev3))))

stopifnot(all.equal(ev1,ev1a), all.equal(ev2,ev2a), all.equal(ev3,ev3a))

## Not run:  #Requires test DB setup.
testcol <- mongo("Messages",
                 url="mongodb://test:secret@127.0.0.1:27017/test")
## Mongodb is the protocol
## user=test, password =secret
## Host = 127.0.0.1 -- localhost
## Port = 27017 -- Mongo default
## db = test
## collection = Messages
testcol$remove('{}')  ## Clear everything for test.

ev1 <- saveRec(ev1,testcol)
ev2 <- saveRec(ev2,testcol)
ev3 <- saveRec(ev3,testcol)
```

```
ev1b <- getOneRec(buildJQuery("_id"=ev1@"_id"),testcol,parseEvent)
ev23 <- getManyRecs(buildJQuery("uid"="Phred"),testcol,parseEvent)
stopifnot(all.equal(ev1,eb1b), length(ev23)==2L)


## End(Not run)
```

`Event-class` *Class* `"Event"`

Description

This class represents a generalize event happening in a simulation or a game. All events have common metadata which identify what happened (`verb`), what was effected (`object`), the time of the action (`timestamp`) as well as the application `app` and user (`uid`). However, the `details` field of the event will differ dependiing on the `app`, `verb`, and `object`.

Objects from the Class

Objects can be created by calls the the `Event` function, although more typically they are passed to the `EIEngine` by the presentation process.

The event object is a simplifcation of the xAPI events (Betts and Ryan, 2018). In particular, in the xAPI format, both the `verb` and `object` fields are complex objects which have a long URL-like identifier to make sure they are unique across applications. In the EIEvent prototocl, only the `app` field is given a long URL-like name. The application should define the acceptable vocabulary for verbs and objects, which should correpsond to the `verb` and `object` fields of the `Rule` objects.

Slots

`verb`: Object of class `"character"` which provides an identifier for the action which just occurred.

`object`: Object of class `"character"` which provides an identifier for the direct object of the action which just occured.

`_id`: Object of class `"character"` the Mongo database identifier; this should not be modified.

`app`: Object of class `"character"` a unique identifier for the application. This defines which `EIEngine` is used to handle the event.

`uid`: Object of class `"character"` identifier for the user (student or examinee).

`context`: Object of class `"character"` an identifier for the context, for applications in which the context is determined by the presentation process.

`sender`: Object of class `"character"` the name of the process which generated the event, usually the presentation process.

`mess`: Object of class `"character"` a title for the message, used by the `P4Message` dispatch system.

`timestamp`: Object of class `"POSIXt"` giving the time at which the event occurred.

`data`: Object of class `"list"` giving the contents of the message. This will be details specific to the verb and object.

Extends

Class "`P4Message`", directly. All fields except `verb` and `object` are inherited from the parent.

Note that the `Event` is the basic message sent from the presentation process to the evidence identification process in the four-process architecture (Almond, Steinberg and Mislevy, 2002). It has been extended slightly, borrowing (and simplifying) the `verb` and `object` header fields from the xAPI format (Almond, Shute, Tingir, and Rahimi, 2018).

Note that `Physics Playground` uses a slightly different architecture. The presentation uses Learning Locker (HT2Labs, 2018) to log events in the xAPI format into a mongo database. When the game level completes, a message is sent to the `EIEngine` which extracts the relevant messages from the Learning Locker database and simplifies them into the event format.

Methods

as.jlist `signature(obj = "Event", ml = "list")`: ...

object `signature(x = "Event")`: Fetches the object component.

verb `signature(x = "Event")`: Fetches the verb component.

show `signature(object = "Event")`: Prints event object.

toString `signature(x = "Event")`: Produces <> representation of object.

all.equal.Event `(target, current, ..., checkTimestamp=FALSE, check_ids=TRUE)`: (S3 method) Checks for equality. The `checkTimestamp` flag controls whether or not the timestamp is checked. The `check_ids` flag controls whether or not the database IDs are checked.

Author(s)

Russell Almond

References

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: `https://education.umd.edu/file/11333/download?token=kmOIVIwi`, Video: `https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4`.

Betts, B, and Smith, R. (2018). The Leraning Technology Manager's Guid to xAPI, Second Edition. HT2Labs Research Report: `https://www.ht2labs.com/resources/the-learning-technology-managers-guide-to-`
`#gf_26`.

HT2Labs (2018). Learning Locker Documentation. `https://docs.learninglocker.net/welcome/`.

See Also

The event class is a subclass of `P4Message`.

Other classes in the EIEvent system: `EIEngine`, `Context`, `Status`, `Rule`.

Methods for working with events: `Event`, `parseEvent`

Examples

```
showClass("Event")
```

`executePredicate` *Executes the predicate of an EIEvent Rule.*

Description

An `Rule` object contains a list of Predicates. The name of each condition is the name of an operator and its value is a list giving the field of the `Status` object to be modified as a name and a new value as the value. For example, `"!incr"=c("state.flags.agentCount"=1)` would increment the agent count flag by one. The value can also be a reference to fields in the `Status` or `Event` object. For example, `"!set"=c("state.observables.trophy" = "event.data.trophy")` would set the value of the trophy observable to the value of the trophy datum in the event.

Usage

```
executePredicate(predicate, state, event)
```

Arguments

`predicate`	A named list of actions: see details.
`state`	An object of class `Status` to be modified
`event`	An object of class `Event` which will be referenced in setting the values.

Details

The predicate of a `Rule` is a list of actions to be taken when the rule is satisfied. Each action has the following form:

!op`=list(`*field*`=`*arg*`,...)`

Here, *field* is an identifier of a field in the `Status` object being modified (the `Event` fields cannot be modified). This is in the dot notation (see `setJS`). The setting operator, *!op* is one of the operators described in Predicates. For the `"!set"` operator, the *arg* is the replacement value or field reference that the field will be used as the replacement value. Most of the other operators use *arg* to modify the value of *field* and then replace the value of *field* with the result. For example, the `"!incr"` operator acts much like the C += operator. The `...` represents additional `field`–`arg` pairs to be set.

The *arg* can be a literal value (either scalar or vector) or a reference to another field in the `Status` or `Event` object using the dot notation. Note that certain operations on timers use the `timestamp` field of the Event to update the timer.

In general, a predicate contains a list of actions. These are executed sequentially, but no guarentees are made about the order.

Finally, one special operator allows for expansion. For the `"!setCall"` operator, *arg* should be the name of a function, with arguments `(name, state, event)`, where `name` is the name of the target field, and state and event are the current state and event objects. The value is set to the value returned.

See Predicates for more details.

Value

The function `executePredicate` returns the modified status object.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. https://docs.mongodb.com/manual/.

See Also

`Rule` describes the rule object and Predicates describes the conditions. Conditions describes the condition part of the rule, and `checkCondition` checks the conditions.

The functions `testPredicate` and `testPredicateScript` can be used to test that rule conditions function properly.

Other classes in the EIEvent system: `EIEngine`, `Context`, `Status`, `Event`, `RuleTable`.

Examples

```
st <- Status("Phred","Level 1",timerNames=c("learningsupports"),
   flags=list(lastagent="lever",agentlist=c("lever"),
              noobj=7,noagents=0),
   observables=list(),
   timestamp=as.POSIXct("2018-12-21 00:01"))

ev <- Event("Phred","test","message",
      timestamp=as.POSIXct("2018-12-21 00:01:01"),
      details=list(agent="lever"))


## Set a flag and an observable.
st1 <- executePredicate(list("!set"=list("state.flags.agent"="ramp",
             "state.observables.trophy"="gold")),st,ev)
stopifnot(getJS("state.flags.agent",st,ev)=="ramp",
          getJS("state.observables.trophy",st,ev)=="gold")

## Set a timer
```

```
st1 <- executePredicate(list("!set"=
                             list("state.timers.learningsupports.time"=
                                   as.difftime(0,units="secs"),
                                  "state.timers.learningsupports.run"=TRUE)),
                        st,ev)
stopifnot(
   timerRunning(st1,"learningsupports",as.POSIXct("2018-12-21 00:01:01")),
   timerTime(st1,"learningsupports",as.POSIXct("2018-12-21 00:01:11"))==
   as.difftime(10,units="secs"))


## Delete fields
st1 <- executePredicate(
  list("!unset"=list("state.flags.noobj"="NA", # Set to NA
                     "state.flags.lastagent"="NULL", # Set to NULL
                     "state.flags.noagents"="Delete")), # Delete it.
   st,ev)
stopifnot(is.na(flag(st1,"noobj")),is.null(flag(st1,"lastagent")),
          is.null(flag(st1,"noagents")))


## Modify fields.
st1 <- executePredicate(
   list("!incr" = list("state.flags.noagents"=1,
                       "state.timers.learningsupports"=
                       as.difftime(1,units="mins"))),
   st,ev)
stopifnot(flag(st1,"noagents")==1,
   timerTime(st1,"learningsupports",as.POSIXct("2018-12-21 00:01:11"))==
   as.difftime(60,units="secs"))

st2 <- executePredicate(
   list("!decr" = list("state.flags.noagents"=1,
                       "state.timers.learningsupports"=
                       as.difftime(30,units="secs"))),
   st1,ev)
stopifnot(flag(st2,"noagents")==0,
   timerTime(st2,"learningsupports",as.POSIXct("2018-12-21 00:01:11"))==
   as.difftime(30,units="secs"))


st1 <- executePredicate(
   list("!mult" = list("state.flags.noobj"=2)),st,ev)
stopifnot(flag(st1,"noobj")==14)
st2 <- executePredicate(
   list("!div"  = list("state.flags.noobj"=2)),st1,ev)
stopifnot(flag(st2,"noobj")==7)

st1 <- executePredicate(
   list("!min"  = list("state.flags.noobj"=5)),st,ev)
stopifnot(flag(st1,"noobj")==5)
st1 <- executePredicate(
   list("!max"  = list("state.flags.noagents"=1)),st,ev)
```

```
stopifnot(flag(st1,"noagents")==1)

## Set operators
st1 <- executePredicate(
   list("!addToSet" =list("state.flags.agentlist"="lever")),st,ev)
stopifnot(flag(st1,"agentlist")=="lever")
st2 <- executePredicate(
   list("!addToSet" =list("state.flags.agentlist"="springboard")),st1,ev)
stopifnot(setequal(flag(st2,"agentlist"),c("lever","springboard")))

st3 <- executePredicate(
   list("!pullFromSet" =list("state.flags.agentlist"="lever")),st2,ev)
stopifnot(flag(st3,"agentlist")=="springboard")


st1 <- executePredicate(
   list("!push"=list("state.flags.objects"="Object 1")), st,ev)
stopifnot(flag(st1,"objects")=="Object 1")

st2 <- executePredicate(
   list("!pop"=list("state.flags.objects"="state.flags.lastObject")),st1,ev)
       #Pop first object off the stack and set lastObject to its value.
stopifnot(flag(st2,"lastObject")=="Object 1",
          length(flag(st2,"objects"))==0L)

myOp <- function(name,state,event) {
  return(getJS("state.flags.noobj",state,event)/
         as.double(getJS("state.timers.learningsupports.time",state,event),
         units="mins"))
}

st1 <- executePredicate(
   list("!setCall"=list("state.flags.value"="myOp")),st,ev)
       #Set value to return value of myOp.
stopifnot(!is.finite(flag(st1,"value")))

stx <- Status("Phred","Level 1",timerNames=c("learningsupports"),
   flags=list(lastagent="lever",agentlist=c("lever"),
              noobj=7,noagents=0),
   observables=list(trophyHall=list(),bankBalance=0),
   timestamp=as.POSIXct("2018-12-21 00:01"))

eve <- Event("001c1","initialized","newMoneyEarned",
             timestamp=as.POSIXct("2019-04-29 15:34:35.761500"),
             details=list(gameLevelId="Down Hill",
                          earningType="goldWin",
                          moneyEarned="20",
                          currentMoney="60"
                          ), context="Down Hill")

stx1 <- executePredicate(
          list("!set"=list("state.observables.bankBalance"=
                           "event.data.currentMoney"),
              "!setKeyValue"=list("state.observables.trophyHall"=
                  list(key="event.data.gameLevelId",
                      value="event.data.earningType"))),stx,eve)
```

flag — *Accessor functions for context objects.*

Description

These are basic accessor functions for the fields of an object of class `Status`. Note that both the `flgs` and `observables` fields of the Status object are named lists. The functions `flag(x,name)` and `obs(x,name)` access a single component of those objects.

Usage

```
flag(x, name)
## S4 method for signature 'Status'
flag(x, name)
flag(x, name) <- value
## S4 replacement method for signature 'Status'
flag(x, name) <- value
obs(x, name)
## S4 method for signature 'Status'
obs(x, name)
obs(x, name) <- value
## S4 replacement method for signature 'Status'
obs(x, name) <- value
## S4 method for signature 'Status'
app(x)
## S4 method for signature 'Status'
timestamp(x)
```

Arguments

`x`	A `Status` object whose fields will be accessed.
`name`	The name of the component for a `flag` or `obs` field.
`value`	The replacement value for the field.

Value

The functions `flag` and `obs` return the named component of the `flags` or `observables` field of `x` respectively. If no component will the given name exists, they return `NULL`.

The functions `app`, `context`, `timestamp` and `oldContext` return the value of the corresponding field of `x`.

The setter methods return the modified `Status` object.

Author(s)

Russell Almond

See Also

Status describes the state object. See context, context<-, and oldContext for other accessor methods shared by context and other objects.

The functions setJS, getJS and removeJS provide mechanisms for accessing the fields of a status object from Rule Conditions and Predicates.

The following functions access the timer fields of the state object. timer, timerTime, timerRunning, setTimer

Examples

```
st <- Status("Phred","Level 0",timerNames="watch",
   flags=list("list"=list("one"=1,"two"="too"),"vector"=(1:3)*10),
   observables=list("numeric"=12.5,char="foo",
                 "list"=list("one"="a","two"=2),"vector"=(1:3)*100),
   timestamp=as.POSIXct("2018-12-21 00:01"))

stopifnot(
  uid(st) == "Phred",
  context(st)=="Level 0",
  oldContext(st)=="Level 0",
  all.equal(flag(st,"list"),list("one"=1,"two"="too")),
  all.equal(flag(st,"vector"),(1:3)*10),
  all.equal(obs(st,"numeric"),12.5),
  all.equal(obs(st,"char"),"foo"),
  all.equal(obs(st,"list"),list("one"="a","two"=2)),
  all.equal(obs(st,"vector"),(1:3)*100),
  all.equal(timestamp(st),as.POSIXct("2018-12-21 00:01"))
)

context(st) <- "Level 1"
stopifnot(
  context(st)=="Level 1",
  oldContext(st)=="Level 0")

stopifnot(is.null(flag(st,"numeric")))
flag(st,"numeric") <- 17L
stopifnot(!is.null(flag(st,"numeric")),
          flag(st,"numeric")==17L)

flag(st,"list")$two <- "two"
stopifnot(all.equal(flag(st,"list"),list("one"=1,"two"="two")))

obs(st,"vector")[2] <- 2
stopifnot(all.equal(obs(st,"vector"),c(100,2,300)))
```

`getJS` *Gets a field from an object in Javascript notation.*

Description

Fields of a `Status` can be accessed using JavaScript notation, e.g., `state.flags.`*field*, `state.observables.`*field*,or `state.timers.`*name*. Similarly, fields of an `Event` can be referenced using `event.verb`, `event.object`, `event.data.`*field*, or `event.timestamp`. The function `getJS` fetches the current value of the referenced field from the state or event object.

Usage

```
getJS(field, state, event)
getJSfield(obj,fieldlist)
```

Arguments

`field`	A character scalar describing the field to be referenced (see details).
`state`	An object of type `Status` giving the current status of the user in the system.
`event`	An object of type `Event` giving the event being processed.
`obj`	A collection object to be accessed. The object implmenting `state.flags`, `state.observables` or `event.details`, or one of sub-components.
`fieldlist`	The successive field names as a vector of characters (split at the '.' and excluding the initial `state.flags`, `state.observables` or `event.details`.

Details

Both the Conditions and Predicates of `Rule` objects need to reference parts of the current `state` and `event`. As these rules are typically written in JSON, it is natural to reference the parts of the `Status` and `Event` objects using javascript notation. Javascript, like R, is a weakly typed language, and so javascript objects are similar to R lists: a named collection of values. A period, '.', is used to separate the object from the field name, similar to the way a '$' is used to separate the field name from the object reference when working with R lists. If the object in a certain field is itself an object, a succession of dots. Thus a typical reference looks like: *object.field.subfield* and so forth as needed.

In EIEvent rules, only two objects can be referenced: `state`, the current `Status`, and `event`, the current `Event`. Therefore, all dot notation field references must start with either `state` or `event`. Furthermore, `Status` and `Event` objects have only a certain number of fields so only those fields can be referenced.

The event object has one field which can contain arbitrary collections, `event.data`. The state object has two `state.flags` and `state.observables`. (The state also contains a collection of timer objects, `state.timers`, which has special rules described below.) Each of these is a named collection (list in R), and components can be refenced by name. The expressions "`event.data.`*name*", "`state.flags.`*name*", and "`state.observables.`*name*" reference an object named *name* in the data field of the event, or the flags or observables field of the state respectively. Note that the available components of these lists fields will depend on the context of the simulation and the verb and object of the event.

The fields of `event.data`, `state.flags` and `state.observables` could also be multipart objects (i.e., R lists). Additional dots can be used to reference the subcomponents. Thus "`event.data.position.x`" references the x-coordinate of the position object in the event data. These dots can be nested to an aribrary depth.

The fields of `event.data`, `state.flags.` and `state.observables.` can also contain unnamed vectors (either character, numeric, or list). In this case square brackets can be used to index the elements by position. Indexes start at 1, as in R. For example, "`state.flags.agentList[3]`" references the third value in the agentList flag of the status. Currently only numeric indexes are allowed, variable references are not, nor can sublists be selected.

The function `getJSfield` is an internal function which is used to access components. It is called recursively to access fields which are themselves lists or vectors.

Value

The value of `getJS` the the contents of the referenced field. If the referenced field does not exist, or the reference is not well formed, then an error is signaled.

Fields of the `Event` object.

The following expressions reference the fields of the `Event` object.

- `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.`*field* The value of the extra data field named *field*.

The following additional fields can also be referenced, but are seldom used. In many cases, these fields are handled by the `EIEngine` before rule processing begins, so their values are irrelevant.

- `event.app` The application ID associated with the current event..
- `event.context` The simulator context of the current event as recorded by the presentation process.
- `event.uid` The user ID of the student or player.
- `event.message` The message sent from the presentation process. (Often something like "New Event".)

Fields of the `Status` object.

The following fields of the `Status` object can be referenced.

- `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.`*field* The value of the observable named *field*.
- `state.timers.`*field* The the timer named *field*. Note that `state.timers.`*field*`.time` or `.value` refers to the current elapsed time of the timer, and `state.timers.`*field*`.run` or `.running` is a logical value which refers to whether or not the timer is running.
- `state.flags.`*field* The value of the observable named *field*.

The following additional fields are also recognized, but again they are primarily for use in the `EIEngine`.

- `state.uid` The user ID of the student or player. (Should be the same of that of the event.)
- `state.timestamp` The timestamp of the last event encorporated into the status.

Timers

The `state.timers` field holds a named list of objects of class `Timer`. These behave as if they have two settable subfields: `running` (or `run`) and `time` (or `value`).

The `running` (or `run`) virtual field is a logical field: `TRUE` indicates running and `FALSE` indicates paused. Setting the value of the field will cause the timer to resume (start) or pause depending on the value.

The `time` (or `value`) field gives the elapsed time of the timer. Setting the field to zero will reset the timer to zero, setting it to another value will adjust the time.

Note

It is clear that some kind of indirect reference (i.e., using variables, either integer or character, inside of the square brackects) is needed. This may be implemented in a future version.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: `https://education.umd.edu/file/11333/download?token=kmOIVIwi`, Video: `https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4`.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. `https://docs.mongodb.com/manual/`.

See Also

The functions `setJS` for setting fields and `removeJS` for removing fields (only allowed with state objects). This function is called from the functions `checkCondition` and `executePredicate`.

The help files Conditions and Predicates each have detailed descriptions of rule syntax.

Other classes in the EIEvent system: `EIEngine`, `Context`, `Status`, `Event`, `Rule`.

Examples

```
st <- Status("Phred","Level 0",timerNames="watch",
   flags=list("list"=list("one"=1,"two"="too"),"vector"=(1:3)*10),
   observables=list("numeric"=12.5,char="foo",
                "list"=list("one"="a","two"=2),"vector"=(1:3)*100),
   timestamp=as.POSIXct("2018-12-21 00:01"))
context(st) <- "Level 1"

ev <- Event("Phred","test","message",
      timestamp=as.POSIXct("2018-12-21 00:01:01"),
      details=list("list"=list("one"=1,"two"=1:2),"vector"=(1:3)))

stopifnot(
getJS("state.context",st,ev)=="Level 1",
getJS("state.oldContext",st,ev)=="Level 0",
getJS("state.observables.numeric",st,ev)==12.5,
getJS("state.observables.char",st,ev)=="foo",
all.equal(getJS("state.observables.list",st,ev),list("one"="a","two"=2)),
getJS("state.observables.list.one",st,ev)=="a",
getJS("state.observables.vector[2]",st,ev)==200,
all.equal(getJS("state.flags.list",st,ev),list("one"=1,"two"="too")),
getJS("state.flags.list.two",st,ev)=="too",
getJS("state.flags.vector[3]",st,ev)==30
)

stopifnot(
getJS("state.timers.watch.running",st,ev)==FALSE,
getJS("state.timers.watch.time",st,ev)==as.difftime(0,units="secs")
)

timerTime(st,"watch",as.POSIXct("2018-12-21 00:01")) <-
      as.difftime(1,units="mins")
timerRunning(st,"watch", as.POSIXct("2018-12-21 00:01")) <- TRUE

stopifnot(
getJS("state.timers.watch.run",st,ev)==TRUE,
getJS("state.timers.watch.value",st,ev)==as.difftime(61,units="secs")
)
## Note that value of a running timer references difference between
## internal start time and current time as recorded by the event.


stopifnot(
getJS("event.verb",st,ev)=="test",
getJS("event.object",st,ev)=="message",
getJS("event.timestamp",st,ev)==as.POSIXct("2018-12-21 00:01:01"),
all.equal(getJS("event.data.list",st,ev),list("one"=1,"two"=1:2)),
all.equal(getJS("event.data.list.two",st,ev),1:2),
getJS("event.data.vector[3]",st,ev)==3
)
```

handleEvent *Does the processing for a new event.*

Description

This function finds the appropriate rules for handling an event then runs them. It is the core function for the `EIEngine`.

Usage

```
handleEvent(eng, event)
processEvent(eng, state, event)
```

Arguments

`eng`	An object of class `EIEngine` which will process the event.
`state`	An object of class `Status` that will be updated by the rules processing the event.
`event`	An object of class `Event` that will be processed.

Details

The function `processEvent` first calls `eng$findRules` to find rules appropriate for the event. If no rules exist, it returns NULL (indicating that the state has not changed and does not need to be saved.). If there are rules, then the following functions are run in sequence:

1. `runStatusRules`: update internal status variables (flags and timers).
2. `runObservableRules`: update external status variables (observables).
3. `runContextRules`: check to see if the context (game level) has changed.
4. `runTriggerRules`: run rules to general messages for other processes if necessary.
5. `runResetRules`: resets the state for the beginning of a new context.

It then updates the timestamp and returns the modified status. If an error occurs during any of the steps, then an object of class `try-error` is returned (see `withFlogging`).

The function `handleEvent` first finds the current status for the user from the `UserRecordSet` linked to the engine. It then runs `processEvent`, and then if necessary it saves the updated status.

Value

The function `processEvent` returns `NULL` if there were no applicable rules, an object of class `Status` if the rules executed correctly and an object of class `try-error` if processing the rules generated an error.

The function `handleEvent` returns the current `Status` if the handling was successful and an object of class `try-error` in an error was generated.

Note

This function uses the `flog.logger` mechanism. The following information is reported at various thresholds:

TRACE
- The results of `checkCondition` are reported.
- The specific rules found from each query are reported.
- A message is logged as each rule is run.

DEBUG
- When an error occurs information about the state, event and rule where the error occurred as well a stack trace are logged.
- Each event is logged as it is processed.
- Rule searches are logged and the number of rules reported.
- A message is logged when each phase starts.
- A message is logged when the context changes.

INFO and above If an error occurs the error is logged along with context information.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: `https://education.umd.edu/file/11333/download?token=kmOIVIwi`, Video: `https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4`.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. `https://docs.mongodb.com/manual/`.

See Also

Conditions and Predicates each have detailed descriptions. The functions `checkCondition` and `executePredicate` run the condition and predicate parts of the rule. The functions `runRule` and `runTRule` run the indivual rules, and the functions `runTriggerRules`, `runStatusRules`, `runObservableRules`, `runResetRules`, and `runContextRules` run sets of rules.

The functions testQuery and testQueryScript can be used to test that rule conditions function properly. The functions testPredicate and testPredicateScript can be used to test that predicates function properly. The functions testRule and testRuleScript can be used to test that rule conditions and predicates function properly together. The class RuleTest stores a rule test.

Other classes in the EIEvent system: EIEngine, Context, Status, Event, RuleTable.

Examples

```
## Not run:
app <- "ecd://epls.coe.fsu.edu/P4test"
loglevel <- "DEBUG"
cleanFirst <- TRUE
eventFile <- "/home/ralmond/Projects/EvidenceID/c081c3.core.json"
## Adjust the path here as necessary
source("/usr/local/share/Proc4/EIini.R")
flog.appender(appender.file(logfile))
flog.threshold(loglevel)

## Setup Listeners
listeners <- lapply(names(EI.listenerSpecs),
                    function (ll) do.call(ll,EI.listenerSpecs[[ll]]))
names(listeners) <- names(EI.listenerSpecs)
if (interactive()) {
  cl <- new("CaptureListener")
  listeners <- c(listeners,cl=cl)
}

## Make the EIEngine
eng <- do.call(EIEngine,c(EIeng.params,list(listeners=listeners),
                          EIeng.common))

## Process Event file if supplied
if (!is.null(eventFile)) {
  system2("mongoimport",
          sprintf('-d
          stdin=eventFile)
  ## Count the number of unprocessed events
  NN <- eng$eventdb()$count(buildJQuery(app=app(eng),processed=FALSE))
}

for (n in 1L:NN) {
  eve <- eng$fetchNextEvent()
  out <- handleEvent(eng,eve)
  eng$setProcessed(eve)
}

## End(Not run)
```

`loadContexts` *Loads a set of contexts from a matrix description*

Description

The easiest way to load a set of `Context` objects is from a matrix of cross references. There are columns corresponding to the fixed fields of the context object (`cid`,`number`, `name` and `doc`). The other columns are logical variables that refer to context sets, which take on a true value when the context represented by the row belongs to the set represented by the column.

Usage

```
loadContexts(conmat, set, app)
```

Arguments

`conmat`	A matrix representing a collection of contexts, see details.
`set`	Either a `ContextSet` object, or a list to which the new contexts will be added.
`app`	A character scalar giving the name of the application; used in constructing the contexts.

Details

A context matrix is a `data.frame` with the following columns: `cid` (character, required), `number` (numeric, required), `name` (character, optional) and `doc` (character, required). The `cid` and `number` fields should also be unique across all the entries as they serve as indexes for the `ContextSet` object. The `name` column if omitted defaults to the `cid` (the difference is that the name is designed to be more human readable). The `doc` if omitted produces an empty documentation string. The remaining columns are logical and are indicators for set membership.

The following is an example context matrix:

	cid	**number**	**name**	**doc**	**Set1**	**Subset1.1**	**Set2**
[,1]	Set1	-100	"Set 1"	"A context set"	0	0	0
[,2]	Subset1.1	-110	"Subet 1.1"	"A subset of Set 1"	1	0	0
[,3]	Set2	-200	"Set 2"	"Another context set"	0	0	0
[,4]	Task1	100	"First Task"	"A task beloning to Set 1"	1	0	0
[,5]	Task1a	101	"First Task Variant"	"A task beloning to Subet 1.1"	1	1	0
[,6]	Task2	200	"Second Task"	"A task beloning to Set 2"	0	0	1

A couple things to note about this matrix. First, both actual contexts and context sets are represented (the are both represented internally by `Context` objects. The names of the extra columns correspond to the names of the context sets. If the database is used to store contexts, it recommended to give sets a negative number and actual contexts positive numbers. If a list is used, then the numbers need to be list indexes, and so should be low numbers.

Also, the `belongsTo` relationship is not transitive (or at least the transitive implications are not computed), so Task1a must have both set and subset memebership defined.

The code loops through the rows of the table, and creates a new context for each row. This is then added to the `set` argument. If an existing context of the same cid and number exists, it is replaced, otherwise a new one is added. If the context is replacing an old one, the old one should have both the same name and number as the replacement. The method for `updateContext` for a `ContextSet` object (called by `loadContexts`) checks for this inconsistency; the list method does not.

The matrix is to represent the entire context set, calling `clearContexts` before loading the matrix should eliminate conflicts.

Value

The modified `set` argument is returned. This is either a `list` or a `ContextSet`.

Note

The `ContextSet` class is a reference class. Therefore, this function destructively modifieds the database associated with the set.

The `list` is an ordinary R functional class, and the function behaves in a functional manner when called with a list as a set.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the context system: `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

See Also

`Context` describes the context object, and `ContextSet` describes the context set object.

The functions `matchContext`, `updateContext` and `clearContexts` are used to manipulate context sets.

Examples

```
conmat <- read.csv(file.path(library(help="EIEvent")$path,"testScripts",
                "SampleContextSheet.csv"))

conmat1 <- conmat
conmat1$Number <- 1:nrow(conmat) ## List style needs small integer indexes.
setlist <- loadContexts(conmat1,list(),"ecd://epls.coe.fsu.edu/rga/test")
stopifnot(all.equal(names(setlist),as.character(conmat$CID)))

testSet <- ContextSet$new(app="ecd://epls.coe.fsu.edu/rga/test",
                      dbname="test",dburi="mongodb://localhost")

clearContexts(testSet)
```

```
loadContexts(conmat,testSet,"ecd://epls.coe.fsu.edu/rga/test")
c1 <- matchContext(28L,testSet)
c2 <- matchContext("Stairs",testSet)
stopifnot(all.equal(c1,c2))
```

`loadRulesFromList` *Functions for loading rules into database.*

Description

These functions load rules into a `RuleTable` (or the database for a `EIEngine`). The function `testAndLoad` also runs the test suite as the rules are loaded.

Usage

```
loadRulesFromList(set, rulelist, stopOnDups = TRUE)
testAndLoad(set, filename, stopOnDups = FALSE)
```

Arguments

`set`	An object of class `RuleTable` into which the rules will be loaded.
`rulelist`	A list of `Rule` objects.
`stopOnDups`	A logical value which controls how names with duplicate values will be treated. If true, then an error will be signaled if the rule to be added has the same name as one in the database. If false, the old rule will be replaced.
`filename`	A file which contains a number of `RuleTest` objects in JSON format (see `testRule`). The tests will be run, and if successful the rules will be extracted and loaded.

Details

The `loadRulesFromList` function is more often called from the `$loadRules()` method of the `EIEngine` class. The easiest way to maintain the rules is by a series of one or more JSON files. These can be read using:

```
rulelist <- lapply(fromJSON("rulefile.json",FALSE), parseRule)
```

With `stopOnDups==FALSE`, the functions will issue warnings when they are replacing rules, especially if the versions are different. However, it is probably easier to first clear out the old rules when replacing the rules. This can be done with the `$clearAll()` method of the `RuleTable` class or the the `$clearAllRules()` method of the `EIEngine` class.

The `testAndLoad` variant is not yet fully tested, but the idea is that the file would be a test script of the kind that could be tested with `testRuleScript`. It will run the tests, and load the rules only if the tests pass. (Note that as a rule could have multiple tests, it will be loaded if any of the tests pass.)

Value

Both of these function return the `set` argument invisibly.

Note

The `testAndLoad` variant is not fully tested.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

There is a "quick start" document which describes how to set up the engine. This is available at https://pluto.coe.fsu.edu/Proc4/EIQuickStart.pdf.

See Also

See `Rule` and `parseRule` for the rule object. See `RuleTable` for the rule collection object and `EIEngine` for the full engine.

Examples

```
## Not run:
rls <- RuleTable$new("ecd://epls.coe.fsu.edu/P4test","test",
                     "mongodb://localhost")
sRules <- lapply(fromJSON(file.path(config.dir, "Rules",
                                    "CombinedRules.json"),FALSE),
                 parseRule)
rls$clearAll()
loadRulesFromList(rls,sRules)

## End(Not run)
```

---

mainLoop — *Runs the* `EIEngine` *as a server.*

Description

The `mainLoop` is used when the `EIEngine` is used as a server. It checks the database, for unprocessed `Event` objects, and calls `handleEvent` on them in the order of their `timestamps`. As a server, this is potentially an infinite loop, see details for ways of gracefully terminating the loop.

Usage

```
mainLoop(eng)
```

Arguments

`eng`	An instance of `EIEngine` which will be used as a server.

Details

The `EIEngine` class uses the Events collection in the database (`eng$eventdb()`) as a queue. All events have a `processed` field which is set to true when the event is processed. Thus the main loop iterates over the following three statements:

1. Fetch the oldest unprocessed Event: `eve <- eng$fetchNextEvent()`.
2. Process the event: `out <- handleEvent(eng,eve)`. (Note: this expression will always return. If it generates an error, the error will be logged and an object of class `try-error` will be returned.)
3. Mark the event as processed: `eng$setProcessed(eve)`.

At its simplest level, the funciton produces an infinite loop over these three statements, with some additional steps related to logging and control.

First, if the event queue is empty, the process sleeps for a time given by `eng$waittime` and then checks the queue again. At the same time, it checks status of the active flag for the process using the `eng$isActivated()` call. By default this checks the `active` field of the record corresponding to `app(eng)` in the collection `AuthorizedApps` in the database `Proc4`. Setting that field to false manually will result in the `mainLoop` terminating when the queue is empty. As R is running in server mode when this happens, this often needs to be done using an external process. The following command issues from the Mongo shell will shut down the server for an application containing the string "appName" as part of its name.

```
db.AuthorizedApps.update({app:{$regex:"appName"}},    {$set:{active:false}});
```

To facilitate testing, the field `eng$processN` can be set to a finite value. This number is decremented at every cycle, and when it reaches 0, the `mainLoop` is terminated, whether or not their are any remaining events to be processed. Setting `eng$processN` to an infinite value, will result in an infinite loop that can only be stopped by using the active flag (or interrupting the process).

Value

There is no return value. The function is used entirely for its side effects.

Note

Currently, when running in server model (i.e., with `eng$processN` set to infinity), there are two ways of stopping the engine: a clean stop after all events are processed using the `active` flag, and an immediate stop, possibly mid cycle, by killing the server process. It became apparent during testing that there was a need for a graceful but immediate stop, i.e., a stop after processing the current event. This should appear in later versions.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

There is a “quick start” document which describes how to set up the engine. This is available at `https://pluto.coe.fsu.edu/Proc4/EIQuickStart.pdf`.

See Also

The class EIEngine describes the setup of the engine, and the function handleEvent describes the processing that occurs for each event.

Examples

```
## Not run:
## From EIEvent.R script
app <- "ecd://epls.coe.fsu.edu/P4test"
loglevel <- "DEBUG"
cleanFirst <- TRUE
eventFile <- "/home/ralmond/Projects/EvidenceID/c081c3.core.json"

## Initialization details (from EIini.R script)
EIeng.common <- list(host="localhost",username="EI",password="secret",
                     dbname="EIRecords",P4dbname="Proc4",waittime=.25)
appstem <- basename(app)
EIeng.params <- list(app=app)
logfile <- file.path("/usr/local/share/Proc4/logs",
                     paste("EI_",appstem,"0.log",sep=""))
EI.listenerSpecs <-
  list("InjectionListener"=list(sender=paste("EI",appstem,sep="_"),
            dbname="EARecords",dburi="mongodb://localhost",
            colname="EvidenceSets",messSet="New Observables"))

## Setup logging
if (interactive()) {
  flog.appender(appender.tee(logfile))
} else {
  flog.appender(appender.file(logfile))
}
flog.threshold(loglevel)
## Setup Listeners
listeners <- lapply(names(EI.listenerSpecs),
                    function (ll) do.call(ll,EI.listenerSpecs[[ll]]))
names(listeners) <- names(EI.listenerSpecs)
if (interactive()) {
  cl <- new("CaptureListener")
  listeners <- c(listeners,cl=cl)
}
## Make the EIEngine
eng <- do.call(EIEngine,c(EIeng.params,list(listeners=listeners),
                          EIeng.common))
## Clean out old records from the database.
if (cleanFirst) {
  eng$eventdb()$remove(buildJQuery(app=app(eng)))
  eng$userRecords$clearAll(FALSE)   #Don't clear default
  eng$listenerSet$messdb()$remove(buildJQuery(app=app(eng)))
  for (lis in eng$listenerSet$listeners) {
    if (is(lis,"UpdateListener") || is(lis,"InjectionListener"))
      lis$messdb()$remove(buildJQuery(app=app(eng)))
  }
}
## Process Event file if supplied
if (!is.null(eventFile)) {
  system2("mongoimport",
          sprintf('-d
          stdin=eventFile)
  ## Count the number of unprocessed events
  NN <- eng$eventdb()$count(buildJQuery(app=app(eng),processed=FALSE))
}
if (!is.null(eventFile)) {
  ## This can be set to a different number to process only a subset of events.
  eng$processN <- NN
}
## Activate engine (if not already activated.)
eng$activate()
mainLoop(eng)

## Depending on value of NN, this may not terminate!

## End(Not run)
```

`matchContext` *Find or replace contexts in a context set*

Description

The function `matchContext` finds a context by string or numeric id. The function `updateContext` adds a new context to a set or replaces an existing one (see details). The function `clearContext` clears all contexts associated with this application.

Usage

```
matchContext(id, set)
## S4 method for signature 'character,ContextSet'
matchContext(id, set)
## S4 method for signature 'numeric,ContextSet'
matchContext(id, set)
## S4 method for signature 'character,list'
matchContext(id, set)
## S4 method for signature 'numeric,list'
matchContext(id, set)
updateContext(con, set)
## S4 method for signature 'Context,ContextSet'
updateContext(con, set)
## S4 method for signature 'Context,list'
updateContext(con, set)
clearContexts(set)
## S4 method for signature 'ContextSet'
```

```
clearContexts(set)
## S4 method for signature 'list'
clearContexts(set)
```

Arguments

id	This should be either a character or numeric scalar giving the cid or number of the desired context.
con	This should be an object of type Context to be added or replaced.
set	This should either be an object of type ContextSet or a list.

Details

The functions are for manipulating collections of contexts. The robust way to implement a context set is to use the ContextSet class, which is an interface to database with dual indexes: one on cid and app and one on number and app. Thus each application maintains its own set of contexts in the same database.

A lightweight implementation can be made using a list with names corresponding to the cid and position in the list corresponding to number. This representation is primarily intended for testing.

The function matchContext is polymorphic on both its arguments. If the first argument is character, it searches based on context id, if the first argument is numeric, it searches based on numeric id (position in the list if the second argument is a list). If no matching context is found, it returns NULL. If the set is a ContextSet successes are logged at the DEBUG level, and failures at the INFO level.

The function updateContext performs a database update operation, that is it replaces the context in the set if it already exists, or adds the context to the set if it does not. If the set is a ContextSet it first checks for an existing context with that character and numeric id. If such a context exists it is replaced; otherwise a new context is added. However, if the new character and numeric ids reference different contexts, then a warning is logged and the contexts are not replaced. If the set is a list, then no checking is done. The context at the given numeric location is replaced and given the new name. This may produce errors or an inconsistent state.

The function clearContexts removes all of the contexts. In the case of the list, it merely returns an empty list. In the case of the ContextSet it removes all contexts with the application id of the set. Other applications should remain untouched.

Value

The function matchContext returns an object of class Context if a class was found and NULL otherwise.

The functions updateContext and clearContexts return the (modified) set argument.

Note

Note that these functions obey functional or reference semantics depending on the set argument. As ContextSet is a reference class, these functions modify the database and hence the results are propagated. However, if set is a list, then normal functional semantics are used, and the resulting modified list needs to be stored in an environment to make the modification permanent.

Author(s)

Russell Almond

See Also

`Context`, `ContextSet`, `loadContexts`

Examples

```
conmat <- read.csv(file.path(library(help="EIEvent")$path,"testScripts",
                 "SampleContextSheet.csv"))

conmat1 <- conmat
conmat1$Number <- 1:nrow(conmat) ## List style needs small integer indexes.
setlist <- loadContexts(conmat1,list(),"ecd://epls.coe.fsu.edu/rga/test")
stopifnot(all.equal(names(setlist),as.character(conmat$CID)))

matchContext(3L,setlist)
matchContext("Stairs",setlist)


testSet <- ContextSet$new(app="ecd://epls.coe.fsu.edu/rga/test",
                      dbname="test",dburi="mongodb://localhost")

clearContexts(testSet)
loadContexts(conmat,testSet,"ecd://epls.coe.fsu.edu/rga/test")
matchContext(28L,testSet)
matchContext("Stairs",testSet)
```

`Predicates` *Functions that modify state when rule is triggered.*

Description

A `Rule` contains both Conditions and Predicates. The latter is a list of operations which are run when the conditions are satisfied. The list of predicates has the form *!op*=`list(`*target*=*arg*`,...)`. Here `target` is a reference to a field in the `Status` object which is modified by applying the *!op* to the current value of the target and the *arg*.

Usage

```
"!set"(predicate, state, event)
"!unset"(predicate, state, event)
"!incr"(predicate, state, event)
"!decr"(predicate, state, event)
"!mult"(predicate, state, event)
"!div"(predicate, state, event)
```

```
"!min"(predicate, state, event)
"!max"(predicate, state, event)
"!addToSet"(predicate, state, event)
"!pullFromSet"(predicate, state, event)
"!push"(predicate, state, event)
"!pop"(predicate, state, event)
"!start"(predicate, state, event)
"!reset"(predicate, state, event)
"!setKeyValue"(predicate, state, event)
"!setCall"(predicate, state, event)
"!send"(predicate, state, event)
"!send1"(predicate, state, event)
"!send2"(predicate, state, event)
modify(predicate, state, event, op)
```

Arguments

`predicate`	This is a list of the form `list(`*target1*=*arg1*`,` *target2*=*arg2*`, ...)`. It names the fields to be modified and the new values (or arguments used to compute the new values).
`state`	A `Status` object representing the current state of the simulation.
`event`	A `Event` object giving details of the current event. Used for dereferencing references in the *arg*.
`op`	A binary argument used to combine the value of the *target* and the *arg* for the `modify` function.

Details

The predicate of a `Rule` is a list of operations. Each operation has the following form:

!op=`list(`*target*=*arg*`,...)`

Here, *target* is an identifier of a field in the `Status` object being modified (targets in the `Event` object cannot be modified). The target field is in the dot notation (see `setJS`). The query operator, *!op* is one of the operations described in the section ‘Predicate Operators’. The *arg* is a value or field reference that will be used in calculating the new value for the target field. In other words, the statement is effectively of the form *target !op arg*. The `...` represents additional `target`–`arg` pairs to be modified.

The *arg* can be a literal value (either scalar or vector) or a reference to another field in the `Status` or `Event` object using the dot notation.

In general, a predicate contains a list of operators and each operator contains a list of *target*=*arg* pairs. These are each executed sequentially; however, the order is not guarenteed. If order is important, use multiple rules with different priorities.

Finally, one special operator allows for expansion. For the `!setCall` operator, *arg* should be the name of a function, with arguments `(name, state, event)`, where `name` is the name of the target field, and state and event are the current state and event objects. The value is set to the value returned.

Value

An object of class `Status` with the target fields modified.

Predicate Operators

The following operators are supported:

`!set` This operator sets the field to the value of the argument. If the field is a flag or observable and does not exist, it is created. If it is a timer and does not exist, an error is signaled.

`!unset` This is the inverse of the `!set` operator. If the *arg* in this expression is `NULL`, or `NA`, then the field will be set to that value. If *arg* is “Delete”, then the field will be removed.

`!incr`, `!decr`, `!mult`, `!div`, `!min`, `!max` For all six of these, the current value of the field is replaced by the result of combining it with the value of the argument. The combination funcitons are ‘+’, ‘-’, ‘*’, ‘/’, `min`, and `max`, respecitively.

`!addToSet`, `!pullFromSet` These operators assume that the value of the field is a vector representing a set. The operator `!addToSet` adds the argument to the set (if it is not already present), and `!pullFromSet` removes the argument (if it is present).

`!setKeyValue` This operators assumes that the value of the field is a named list representing an association (dictionary). The operator `!setKeyValue` expects its argument to have two fields: `key:` and `value:`. The association `key=value` is added to the set.

`!push` This assumes that the field is a vector which represents a stack. The new value is pushed onto the front of the stack.

`!pop` This assumes that the field is a vector which represents a stack. The first value is removed from the stack. If the argument is a field reference, the field referenced in the argument is set to the popped value. If the arugment is numeric, then that many values are popped off the stack.

`!start`, `!reset` In both cases, the field referenced should be a timer. With no argumets, the `!start` operator sets the time value to zero and sets the timer running and `!reset` sets the timer to zero and does not set it running. In both cases, if the timer of the name specified in the *field* does not exist, one is created. If either is given a logical argument, then the timer is set to running or not according to the argument, overriding the default behavior. If the operator is given a numeric or `difftime` argument, then the timer is set to that time. Finally, a argument which is a list with both a `time` (difftime value) and `running` (logical value) will put the timer in that state.

`!send`, `!send1`, `!send2` Special predicate for trigger rules which builds messages to be sent to various listeners. The versions with the numbers are to allow for multiple messages (JSON complains if there are two fields with the same key.) See `buildMessages`.

`!setCall` This is a trap door which allows for arbitrary R code to be used to calculate the value of the *field*. The argument should be the name of a function with three arguments, the name of the field, the status and the event. The field will be set to the value returned by the function.

Setting Timers

Note that timers (see `Timer`) are treated specially. Each timer has a `.run` (or `.running`) subfield which is true if the timer is running and false if it is paused. It also has a `.value` (or `.time`) field which represents the elapsed time.

Timers can be set using the `!set` operator modifies the state of the timer. Setting the `.run` or `.running` subfield of the timer to a logical value will cause the timer to pause (`FALSE`) or resume (`TRUE`). Setting setting the the `.time` or `.value` will set the elapsed time. Similarly, the `!incr`, `!decr`, etc. operators can be used to change the time value.

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 `POSIXt` 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.

Expansion Mechanisms

The special "!setCall" form obviously allows for expansion. It is particularly designed for value calculations which involve multiple fields in a complex fashion.

It is also possible to expand the set of *!op* functions. The method used for dispatch is to call `do.call(op,list(predicate, state, event))` where *predicate* is a list of *target*=*arg* pairs.

The `modify` function is a useful tool for building new predicates. It combines the current value of the field with the value of the arg using a specified operator. This is used to implement many of the existing operators.

Predicate Testing

The function `checkCondition` is used internally to check when a set of conditions in a rule are satisfied.

The functions `testPredicate` and `testPredicateScript` can be used to test that predicates function properly. The functions `testRule` and `testRuleScript` can be used to test that rule conditions and predicates function properly together.

Note

Don't confuse the '!' operator with the character "!" used at the start of the predicate operator names.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. https://docs.mongodb.com/manual/.

See Also

The special `!send` predicate (and `buildMessages` is documented separately).

`Rule` describes the rule object and Conditions describes the Conditions.

The functions `testRule` and `testRuleScript` can be used to test that rule conditions and predicates function properly together.

Other classes in the EIEvent system: `EIEngine`, `Context`, `Status`, `Event`, `RuleTable`.

Examples

```
list (
## Set a flag and an observable.
"!set"=list("state.flags.agent"="ramp",
            "state.observables.trophy"="gold"),
## Set a timer
"!set"=list("state.timers.learningsupports.time"=as.difftime(0,units="secs"),
            "state.timers.learningsupports.run"=TRUE),
## Delete fields
"!unset"=list("state.flags.agent"="NA", # Set to NA
              "state.flags.slider"="NULL", # Set to NULL
              "state.flags.unneeded"="Delete"), # Delete it.

## Modify fields.
"!incr" = list("state.observables.objects"=1), # Add one to objects
"!decr" = list("state.flags.helpuse"=1,  # Subtract 1 from help use.
               "state.timers.learningsupport"=as.difftime(1,units="mins")),
               # Subract one minute from the learning support timer.
"!mult" = list("state.flags.value"=2),# Double value.
"!div"  = list("state.flags.value"=2), # Halve value.
"!min"  = list("state.flags.attempts"=5), # Attempts is less than 5
"!max"  = list("state.flags.attempts"=0), # Attempts is at least 0

## Set operators
"!addToSet" =list("state.flags.agents"="lever"), #Add lever to list of agents
"!pullFromSet" =list("state.flags.agents"="lever"),
             #Remove level from agent list
"!push"=list("state.flags.objects"="Object 1"), #Put Object 1 on the stack.
"!pop"=list("state.flags.objects"="state.flags.lastObject"),
       #Pop first object off the stack and set lastObject to its value.

"!setKeyValue"=
   list("state.observables.trophyHall"=
     list("key"="event.data.gameLevelId","value"="gold")),
## Send operator
"!send" = list("mess"="Money Earned",
               "context"="state.oldContext",
               "data"=
                  list("trophyHall"="state.observables.trophyHall",
                       "bankBalance"="state.observables.bankBalance")),
## Can't use the same operator twice in the same call.
"!send1"="list("mess"="New Observables",
              "context"="state.oldContext",
               "data"=list()),   ## All Observables



## Expansion Operator.
"!setCall"=list("state.flags.value"="myOp"))
       #Set value to return value of myOp.
```

---

queryResult *Accessor Functions for RuleTest objects.*

Description

These are accessor functions for the components of the `RuleTest` class. These allow extraction but not setting of the components.

Usage

```
queryResult(x)
initial(x)
final(x)
event(x)
rule(x)
```

Arguments

x	A `RuleTest` object whose fields are to be accessed.

Value

The function `queryTest` returns a logical value indicating whether or not the `Conditions` of the rule should return true for this test case.

The function `rule` returns the `Rule` being tested.

The function `event` returns the `Event` being processed in the test.

The function `initial` returns the initial `Status` of the system before the test.

The function `final` returns the final `Status` of the system after the test is applied.

Author(s)

Russell Almond

See Also

RuleTest describes the rule test class and RuleTest the constructor function.

Examples

```
testr <- RuleTest(
  name="Simple set",
  doc="Demonstrate predicate test mechanism.",
  initial = Status("Fred","test",
    timestamp=as.POSIXct("2018-12-21 00:01:01")),
  event= Event("Fred","test","rule",details=list(agent="ramp"),
    timestamp=as.POSIXct("2018-12-21 00:01:01")),
  rule=Rule(predicate=list("!set"=c("state.observables.rampused"=TRUE)),
         ruleType="Observable"),
  queryResult=TRUE,
  final=Status("Fred","test",observables=list("rampused"=TRUE),
    timestamp=as.POSIXct("2018-12-21 00:01:01")))

name(testr)
doc(testr)
initial(testr)
event(testr)
rule(testr)
queryResult(testr)
final(testr)
```

---

removeJS *Removes a field from a state object.*

Description

Fields of a Status can be accessed using JavaScript notation, e.g., `state.flags.`*field*, `state.observables.`*field*,or `state.timers.`*name*. The function `removeJS` sets removes the field. This function called when the !unset operator is called with the `"Delete"` argument.

Usage

```
removeJS(field, state)
removeJSfield(target, fieldlist)
```

Arguments

`field`	A character scalar describing the field to be removed (see details).
`state`	An object of type `Status` giving the current status of the user in the system; this argument will be modified.
`target`	A collection object to be accessed. The object implmenting `state.flags`, `state.observables` or `state.timers`, or one of sub-components.
`fieldlist`	The successive field names as a vector of characters (split at the '.' and excluding the initial `state.flags`, `state.observables` or `event.details`.

Details

The Predicates of `Rule` objects update parts of the current `state`. As these rules are typically written in JSON, it is natural to reference the parts of the `Status` objects using javascript notation. Javascript, like R, is a weakly typed language, and so javascript objects are similar to R lists: a named collection of values. A period, '.', is used to separate the object from the field name, similar to the way a '$' is used to separate the field name from the object reference when working with R lists. If the object in a certain field is itself an object, a succession of dots. Thus a typical reference looks like: *object.field.subfield* and so forth as needed.

In EIEvent rules, only the `state`, the current `Status`, can be modified. Therefore, in the predicate all dot notation field references start with `state` Furthermore, only elements of the collection fields of the `Status` (`flags`, `observables` and `timers`) can be referenced. The expressions "`state.flags.`*name*", and "`state.observables.`*name*" reference an object named *name* in the flags or observables field of the state respectively.

The fields `state.flags` and `state.observables` could also be multipart objects (i.e., R lists). Additional dots can be used to reference the subcomponents. Thus "`state.flags.position.x`" references the x-coordinate of the position object in the flag field. These dots can be nested to an aribrary depth. The function `removeJSfield` is a helper function used to remove components of nested items.

The `removeJS` method is called by `executePredicate` when the argument to `!unset` is not `NULL` or `NA`.

Value

The `setJS` function always returns the modified state object. The `setJSfield` function returns the modified colleciton object, or if the fieldlist is empty, `NULL`.

Note

It is clear that some kind of indirect reference (i.e., using variables, either integer or character, inside of the square brackects) is needed. This may be implemented in a future version.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. https://docs.mongodb.com/manual/.

See Also

The functions getJS for accessing fields and setJS for setting fields (only allowed with state objects). This function is called from executePredicate.

The help files Conditions and Predicates each have detailed descriptions of rule syntax.

Other classes in the EIEvent system: EIEngine, Context, Status, Event, Rule.

Examples

```
st <- Status("Phred","Level 0",timerNames="watch",
   flags=list("list"=list("one"=1,"two"="too"),"vector"=(1:3)*10,
              "char"="hello"),
   observables=list("list"=list("one"="one","two"=2),"vector"=(1:3),
             "char"="bar"),
   timestamp=as.POSIXct("2018-12-21 00:01"))

st1 <- removeJS("state.flags.char",st)
stopifnot(all("char"!=names(st1@flags)))

st1 <- removeJS("state.flags.list.two",st)
stopifnot(length(flag(st1,"list"))==1L)

st1 <- removeJS("state.observables.char",st)
stopifnot(is.null(obs(st1,"char")))

st1 <- removeJS("state.observables.list.one",st)
stopifnot(length(obs(st1,"list"))==1L)

st1 <- removeJS("state.timers.watch",st)
stopifnot(is.null(timer(st1,"watch")))
```

Rule *Constructor for EIEvent Rule Objects*

Description

The Rule funciton is the constructor for the Rule object. As Event objects are usually read from a database or other input stream, the parseRule function is recreates an event from a JSON list.

Usage

```
Rule(context = "ALL", verb = "ALL", object = "ALL", ruleType = c("Status", "Observable", "Context", "Tri
parseRule(rec)
## S4 method for signature 'Rule,list'
as.jlist(obj, ml, serialize = TRUE)
```

Arguments

context	A character string describing the task, item or game level during which the event occurred. This should be the name of an object of class Context and it could reference a context set or the special keyword "ALL".
verb	A character scalar identifying the action for which the rule is appropriate: could be the special keyword "ALL".
object	A character scalar identifying the direct object for which the rule is appropriate. Could be the keyword "ALL".
ruleType	A character identifier indicating which phase the rule should be run in. This should be one of the values "State", "Observable", "Context", "Trigger", or "Reset". See the 'Rule Type' section of Event for a description of the phases.
priority	A numeric value indicating the order in which the rules should be run, lower number running earlier in the sequence. It is recommended to use 5 for typical values and smaller numbers for rules which must run earlier, and higher numbers for rules which must run later.
doc	A character vector describing the rule in human language.
name	A character scalar giving an identifier fro the rule. Primarily used in error reporting, debugging and rule management.
condition	A list in a special Conditions syntax. This tests the state of the current Status and Event. If the test returns true, then the predicate is executed.
predicate	A list in a special Predicates syntax. This describes the changes that are made to the Status object (or other actions) that are taken when the rule is triggered.
app	A character scalar providing a unique identifier for the application (game or assessment). This defines the available vocabulary for verb and object, as well as the set of applicable Rule objects.
rec	A named list containing JSON data.
obj	An object of class Event to be encoded.

ml	A list of fields of obj. Usually, this is created by using attributes(obj).
serialize	A logical flag. If true, serializeJSON is used to protect the data field (and other objects which might contain complex R code.

Details

Most of the details about the Rule object, and how it works is documented under Rule-class.

The function as.jlist converts the obj into a named list. It is usually called from the function as.json.

The parseRule function is the inverse of as.jlist applied to an event object. It is designed to be given as an argument to getOneRec and getManyRecs.

Soon to come. A loadRule function which will load in rules. These operate on JSON files, usually part of a test suite.

Value

The functions Rule and parseRule return objects of class event. The function as.jlist produces a named list suitable for passing to toJSON.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. https://docs.mongodb.com/manual/.

See Also

The rule object is described in Rule and Conditions and Predicates each have detailed descriptions. The functions testQuery and testQueryScript can be used to test that rule conditions function properly. The functions testPredicate and testPredicateScript can be used to test that rule conditions function properly. The functions testRule and testRuleScript can be used to test that rule conditions and predicates function properly together. The class RuleTest stores a rule test.

parseMessage and as.json describe the JSON conversion system.

The functions getOneRec and getManyRecs use parseEvent to extract events from a database.

Other classes in the EIEvent system: EIEngine, Context, Status, Event, RuleTable.

Examples

```
r1 <- Rule(name="Coin Rule",
           doc="Set the value of the badge to the coin the player earned.",
           app="ecd://coe.fsu.edu/PPtest",
           verb="satisfied", object="game level",
           context="ALL",
           ruleType="Observable", priority=5,
           condition=list("event.data.badge"=c("silver","gold")),
           predicate=list("!set"=c("state.observables.badge"=
                                   "event.data.badge")))

r1a <- parseRule(ununboxer(as.jlist(r1,attributes(r1))))
all.equal(r1,r1a)
```

Rule-class *Class* "Rule"

Description

A Rule is a declarative predicate for processing events. It has two parts, a condition and a predicate. If the current Status and Event statisfy the condition, then the Status is updated according to the instructions in the Predicate.

Objects from the Class

Objects can be created by calls of the function Rule, or read from JSON using parseRule. Generally rules are stored in a RuleTable.

When the EIEngine gets a new Event, then the applicable rules from the appropriate rule table are fetched. For each rule, if the condition is matched then the predicate is executed.

Slots

_id: Object of class "character" which provides the Mongo database ID of the rule. This generally should not be modified.

app: Object of class "character" giving the name of the application. This should match the RuleTable the rule belongs to.

name: Object of class "character" giving a human readable identifier for the rule; used in error reporting and rule management.

doc: Object of class "character" giving a human readable description of the intent of the rule.

context: Object of class "character" giving the character ID of of a Context or context group to which the rule applies. The special value “ANY” can be used to indicate that the rule applies to all contexts.

`verb`: Object of class `"character"` giving the value of the `verb(Event)` to which the rule applies. The special value “ANY” is used to indicate that the rule applies to all verbs.

`object`: Object of class `"character"` giving the value of the `object(Event)` to which the rule applies. The special value “ANY” is used to indicate that the rule applies to all verbs.

`ruleType`: Object of class `"character"` giving the type of the rule. See the ‘Rule Types’ section.

`priority`: Object of class `"numeric"` providing a partial ordering on the execution sequence of rules. See ‘Rule Applicability and Sequencing’ section.

`condition`: Object of class `"list"` which provides a test for when the rule runs. This should be expressed as a set of restrictions on the fields of the `Status` and `Event` classes. The syntax for this field is described in the ‘Rule Condition’ section.

`predicate`: Object of class `"list"` which describes the action to be performed modifying the state of the `Status` object if the condition is satisfied. The syntax for this field is described in the ‘Rule Condition’ section.

Rule Types

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.
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.

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

The `EIEngine` runs these rules in sequence.

First the status rules are run followed by the observable rules. Although it is suggested that status rules be used to change flags and timers and observable rules to change observables (see `Status`), these rules are not strictly enforced. It is however, guarenteed that status rules will be run before observable rules so that the status rule can be used to calculate intermediate variables which are used in calculating the final observables.

After the observables have been updated, the context rules are run to find out if the context is changed. The context rules are run until the value of `context(Status)` is not equal to the value of `oldContext(Status)`.

Next the trigger rules are run. These cause the `EIEngine` to send a `P4Message` to registered listeners. The primary use is to inform the evidence accumulation engine that new observables are available.

Finally, if `oldContext(Status)` is not equal to `context(Status)`, the reset rules are run to reset the values of timers and counters for the new context type. After this time, the value of `oldContext(Status)` is set to `context(Status)` and the `EIEngine` is ready to process the next event.

Rule Selection and Sequencing

The `EIEngine` applies the rules in five rounds according to the rule types. In each round, the `RuleTable` is searched to find all applicable rules. A rule is applicable if all of the following conditions are met:

1. The value of `ruleType(Rule)` matches the current round.
2. The value of `verb(Rule)` is equal to the value of `verb(Event)` or to the special value "ANY".
3. The value of `object(Rule)` is equal to the value of `object(Event)` or to the special value "ANY".
4. The value of `object(Rule)` is equal to the value of `context(Status)`, is equal to the name of a group context to which `context(Status)` belongs (see `applicableContexts`), or is equal to the special value "ANY".

Any rule which satisfies these four conditions is consider applicable. The `EIEngine` checks the `condition` of all applicable rules, and if the condition is true runs the `predicate`. The exception is the context rule round, in which the rules are run until the first time the condition is satisfied (or more precisely until the value of `context(Status)` changes).

Note that the sequence of rules within a round is arbitrary. In some cases, there will be order dependencies among rules run during the same round. The `priority` field of the rules is used to resolve potential conflicts of this sort. The `EIEngine` sorts the rules by priority before checking them. Rules with lower priority are always run before rules with higher priority, but the sequence of rules with the same priority is arbitrary.

Referencing fields of the `Status` and `Event` objects.

Both conditions and predicates need to reference fields in the `Status` and `Event` objects. This is done using strings which use the dot notation (similar to javascript referencing in JSON documents) to reference fields in the two objects. Thus, `event.data.`*field* references a field named "field" in the `details(Event)`. The following table gives useful dot notation references:

- `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.`*field* The value of the observable named *field*.
- `state.timers.`*field* The the timer named *field*. Note that `state.timers.`*field*`.time` or `.value` refers to the current elapsed time of the timer, and `state.timers.`*field*`.run` or `.running` is a logical value which refers to whether or not the timer is running.
- `state.flags.`*field* The value of the observable named *field*.
- `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.`*field* The value of the extra data field named *field*.

In each of these cases, *field* refers to the name of a field in one of the collections in the `Event` or `Status` object being processed. If the referenced field is a list, then the if the field reference is of the form *field*`.`*component* then the named component of the list is referenced. If the list structure itself contains lists as elements, then multiple '.'s can be used to reference the nested fields. In this respect, the '.' operator performs similarly to the S '$' operator.

If the field references a character or numeric vector, then the '[]' operator can be used to reference elements of that vector. Thus `status.flags.agents[3]` references the third element of a vector called 'agents' found in the flags collection of the status.

The functions `getJS` and `setJS` are used to access the fields, and the help for those functions contains a number of examples.

Rule Conditions

The syntax for the condition part of the rule resembles the query lanugage used in the Mongo database (MongoDB, 2018). There are two minor differences: first the syntax uses R lists and vectors rather than JSON objects and second the '$' operators are replaced with '?' operators.

In general, each element of the list should have the form *field*`=c(`*?op*`=`*arg*`)`. In this expression, *field* references a field of either the `Status` or `EIEngine` (see sQuoteDot Notation section above), *?op* is one of the test operators below, and the argument *arg* is a litteral value (which could be a list) or a character string in dot notation referencing a field of either the `Status` or `Event`. If *?op* is omitted, it is taken as equals if *arg* is a scalar and *?in* if value is a vector. For more complex queries where `arg` is a more complex expresison, the `c()` function is replaced with `list()`.

See Conditions for a list of supported condition operators.

Rule Predicates

The syntax for the predicate of the rule resembles the database update operations used in the Mongo database (MongoDB, 2018). There are two minor differences: first the syntax uses R lists and vectors rather than JSON objects and second the '$' operators are replaced with '!' operators.

The general form of a predicate expression is *!op*`=list(`*field*`=`*arg*`)`. Here *!op* is one of the operations described below, *field* is the name of a field in the `Status` object, and the argument *arg* is either a literal value or a character scalar giving the name of a field of either the `Status` or `Event` in dot notation.

See Predicates a list of supported operations and more information about predicate handling.

Rule Testing

The functions `testQuery` and `testQueryScript` can be used to test that rule conditions function properly.

The functions `testPredicate` and `testPredicateScript` can be used to test that rule conditions function properly.

The functions `testRule` and `testRuleScript` can be used to test that rule conditions and predicates function properly together.

Methods

as.jlist `signature(obj = "Rule", ml = "list")`: helper function for coverting the rule into a JSON object, see `as.json`.

condition `signature(x = "Rule")`: Returns a list giving the conditions for the rule.

app `signature(x = "Rule")`: Returns a character scalar giving the ID of the application this rule belongs to.

context `signature(x = "Rule")`: Returns a character scalar giving the ID of the context or context group to which the rule is applicable.

object `signature(x = "Rule")`: Returns a character scalar giving the object to which the rule is applicable.

predicate `signature(x = "Rule")`: Returns a list giving the predicate for the rule.

ruleType `signature(x = "Rule")`: Returns a character scalar giving the type of the rule.

predicate `signature(x = "Rule")`: Returns a number giving the priority for the rule; lower numbers are higher priority.

verb `signature(x = "Rule")`: Returns a character scalar giving the verb to which the rule is applicable.

show `signature(object = "Rule")`: Prints rule object.

toString `signature(x = "Rule")`: Produces <> representation of object.

all.equal.Rule `(target, current, ..., checkTimestamp=FALSE, check_ids=TRUE)`: (S3 method) Checks for equality.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: `https://education.umd.edu/file/11333/download?token=kmOIVIwi`, Video: `https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4`.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. `https://docs.mongodb.com/manual/`.

See Also

Conditions and Predicates each have detailed descriptions. The functions `checkCondition` and `executePredicate` run the condition and predicate parts of the rule. The functions `runRule` and `runTRule` run the indivual rules, and the functions `runTriggerRules`, `runStatusRules`, `runObservableRules`, `runResetRules`, and `runContextRules` run sets of rules.

The functions `testQuery` and `testQueryScript` can be used to test that rule conditions function properly. The functions `testPredicate` and `testPredicateScript` can be used to test that predicates function properly. The functions `testRule` and `testRuleScript` can be used to test that rule conditions and predicates function properly together. The class `RuleTest` stores a rule test.

Other classes in the EIEvent system: `EIEngine`, `Context`, `Status`, `Event`, `RuleTable`.

Methods for working with Rules: `Rule`, `parseRule`, `ruleType`, `priority`, `condition`, `predicate`, `verb`, `object`, `context`, `name`, `doc`

Examples

```
showClass("Rule")
```

`RuleTable-class` *Class* `"RuleTable"`

Description

This is a containiner for a set of rules for an application.

Extends

All reference classes extend and inherit methods from `"envRefClass"`.

This class extends by containment the `MongoDB` class where the actual rules are stored.

Tracing

The EIEngine uses the `flog.logger` system. In particular, setting the threshold for the “EIEvent” logger, will change the amount of information sent to the log file.

In particular, DEBUG level logging will cause the findRules() method to tell how many rules were returned, and the TRACE level will list them.

Fields

`app`**:** Object of class `character` providing applicaiton ID of the application.

`dbname`**:** Object of class `character` giving the name of the database used.

`dburi`**:** Object of class `character` giving the URI of the database used.

`db`**:** Object of class `MongoDB` giving the actual mongo collection. This should be accessed using the `ruledb()` method to make sure that it has been properly updated.

`stoponduplicate`**:** Object of class `logical`. If true, an error will be signaled if a rule is added with a duplicate name. If false, it will be quietly replaced.

Methods

`updateRule(rule)`: This command will add or replace a rule in the database. If the rule has the same name as an existing rule, the the behavior will depend on the value of `stoponduplicate`. The that flag is FALSE, then the rule will be replaces, if true and error will be logged and the rule not replaced. Note that if the rule has a database ID, and it does not match the ID of the rule it is replacing, an error will be logged and the rule not replaced.

`findRuleByName(name)`: This searchers for a rule by the name filed.

`findRules(verb, object, context, phase)`: This is the key function that finds rules based on `Event` and `Status`. This will return a (possibly empty) list of all matching rules.

`skipDuplicate(newval)`: If called with no arguments, gives the rucrrent value of `stoponduplicate`. If a logical value is given, then the value of the flag is set.

`ruledb()`: This returns the database collection If the database colleciton is currently null, it is inialized. This method should be used rather than raw calls to the db field.

`initialize(app, dbname, dburi, db, stoponduplicate, ...)`: This initialization method.

`clearAll()`: This removes all rules for this application from the database.

Note

A different set of rules is associated with each application ID.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

See Also

The `Rule` class object describes rules. The function `loadRulesFromList` and `testAndLoad` can be used to load rules into the rule table.

The primary classes in the EIEvent system are: `EIEngine`, `Context`, `Status`, `Event`, `Rule`.

The `EIEngine` class is a container for the following classes: `UserRecordSet`, `RuleTable`, `ContextSet`, `TestSet` and `ListenerSet`,

`flog.logger`

Examples

```
showClass("RuleTable")
```

RuleTest *Constructor for EITest or Rule Test.*

Description

These are the constructors for the EITest and RuleTest objects. As test objects are usually read from a database or other input stream, the parseEITest and parseRuleTest functions recreate test objects from a JSON list and as.jlist encodes them into a list to be saved as JSON.

Usage

```
EITest(name, doc = "", app = "default", initial, event, final)
RuleTest(name = paste("Test of Rule", rule), doc = "", app = "default", initial, event, rule, queryResul
parseEITest(rec)
parseRuleTest(rec)
## S4 method for signature 'EITest,list'
as.jlist(obj, ml, serialize = TRUE)
## S4 method for signature 'RuleTest,list'
as.jlist(obj, ml, serialize = TRUE)
```

Arguments

name	A character string identifying the test. Used in Logging.
doc	A character string providing a description of the test.
app	A character string identifying the application.
initial	A Status object giving the initial state of the system.
event	A Event object giving the triggering event.
rule	A Rule object giving rule being tested.
queryResult	A logical value indicating whether or not the Conditions of the rule are satisfied; that is, whether or not the Predicates of the rule should be run.
final	A Status object giving the final state of the system.
rec	A named list containing JSON data.
obj	An object of class RuleTest to be encoded.
ml	A list of fields of obj. Usually, this is created by using attributes(obj).
serialize	A logical flag. If true, serializeJSON is used to protect the data field (and other objects which might contain complex R code).

Details

Most of the details about the EITest and RuleTest objects is documented under the class page.

The function as.jlist converts the obj into a named list. It is usually called from the function as.json.

The parseEITest function is the inverse of as.jlist applied to an EITest object, and parseRuleTest is for a RuleTest object. They are designed to be given as an argument to getOneRec and getManyRecs.

Value

The functions EITest and parseEITest return objects of class EITest. The functions RuleTest and parseRuleTest return objects of class RuleTest. The function as.jlist produces a named list suitable for passing to toJSON.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the context system: https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

See Also

EITest describes the EITest object. RuleTest describes the RuleTest object.

The functions testQuery, testPredicate, and testRule are used to actually execute the tests.

parseMessage and as.json describe the JSON conversion system.

The functions getOneRec and getManyRecs use parseStatus to extract events from a database.

Examples

```
etest <- EITest(
  name="My First test",
  doc="Demonstrate EI Test mechanism.",
  initial = Status("Fred","test",timestamp=as.POSIXct("2018-12-21 00:01:01")),
  event= Event("Fred","test","rule",details=list(trophy="gold"),
         timestamp=as.POSIXct("2018-12-21 00:01:01")),
  final = Status("Fred","test",
     observables=list("trophy"="gold"),
     timestamp=as.POSIXct("2018-12-21 00:01:01")),
  )

el <- as.jlist(etest,attributes(etest))
etesta <- parseEITest(el)
stopifnot(all.equal(etest,etesta))

test <- RuleTest(
  name="Simple test",
  doc="Demonstrate test mechanism.",
  initial = Status("Fred","test",timestamp=as.POSIXct("2018-12-21 00:01:01")),
  event= Event("Fred","test","rule",details=list(trophy="gold"),
         timestamp=as.POSIXct("2018-12-21 00:01:01")),
  rule=Rule(condition=list("event.data.trophy"="gold"),
            predicate=list("!set"=c("state.observables.trophy"="event.data.trophy")),
            ruleType="Observable"),
  queryResult=TRUE,
  final = Status("Fred","test",
      observables=list("trophy"="gold"),
      timestamp=as.POSIXct("2018-12-21 00:01:01")),
  )

tl <- as.jlist(test,attributes(test))
testa <- parseRuleTest(tl)
stopifnot(all.equal(test,testa))


test1 <- RuleTest(
  name="Simple test",
  doc="Demonstrate test mechanism.",
  initial = Status("Fred","test",timestamp=as.POSIXct("2018-12-21 00:01:01")),
  event= Event("Fred","test","rule",details=list(trophy="silver"),
         timestamp=as.POSIXct("2018-12-21 00:01:01")),
  rule=Rule(condition=list("event.data.trophy"="gold"),
            predicate=list("!set"=c("state.observables.trophy"="event.data.trophy")),
            ruleType="Observable"),
  queryResult=TRUE,
  final = Status("Fred","test",
      observables=list("trophy"="silver"),
      timestamp=as.POSIXct("2018-12-21 00:01:01")),
  )

test1a <- parseRuleTest(as.jlist(test1,attributes(test1)))
stopifnot(all.equal(test1,test1a))

testr <- RuleTest(
  name="Simple set",
  doc="Demonstrate predicate test mechanism.",
  initial = Status("Fred","test",
    timestamp=as.POSIXct("2018-12-21 00:01:01")),
  event= Event("Fred","test","rule",details=list(agent="ramp"),
    timestamp=as.POSIXct("2018-12-21 00:01:01")),
  rule=Rule(predicate=list("!set"=c("state.observables.rampused"=TRUE)),
         ruleType="Observable"),
  queryResult=TRUE,
  final=Status("Fred","test",observables=list("rampused"=TRUE),
    timestamp=as.POSIXct("2018-12-21 00:01:01")))

testra <- parseRuleTest(as.jlist(testr,attributes(testr)))
stopifnot(all.equal(testr,testra))


testt <- RuleTest(
  name="Simple message",
  doc="Demonstrate trigger test mechanism.",
  initial = Status("Fred","test",
    timestamp=as.POSIXct("2018-12-21 00:01:01"),
    observables=list(badge="silver")),
  event= Event("Fred","level","finished",details=list(badge="silver"),
    timestamp=as.POSIXct("2018-12-21 00:01:01")),
  rule=Rule(predicate=list("!send"=list()),
         ruleType="Trigger"),
  queryResult=TRUE,
  final=P4Message(uid="Fred",mess="Observables Available",
    details=list(badge="silver"),context="test",sender="EIEvent",
    timestamp=as.POSIXct("2018-12-21 00:01:01")))

testtl <- as.jlist(testt,attributes(testt))
testta <- parseRuleTest(testtl)
stopifnot(all.equal(testt,testta))
```

RuleTest-class   *Class* "RuleTest"

Description

A rule along with a test case for verifying the rule. This is a special case of the more general `EITest`.

Objects from the Class

A rule test object consists of a `Rule` object, plus a test case for the rule. The test case has an *initial* `Status`, a triggering `Event`, and an expected *result* `Status`. In the case of testing trigger rules, the result could be a `P4Message` or list of messages. It also has a logical `queryResult` field which describes whether or not the rule's `Conditions` is satisfied in the test case.

Objects can be created by calls of the form `RuleTest(...)`, or from JSON through `parseRuleTest`.

Slots

`_id`: Object of class `"character"`, the internal mongo identifier.

`app`: Object of class `"character"`, the unique identifier for the application to which this belongs.

`name`: Object of class `"character"`, a human readable name for the test, used in reporting.

`doc`: Object of class `"character"`, a documentation string describing the test.

`initial`: Object of class `Status` which describes the initial state of the system before the test.

`event`: Object of class `Event` which describes the incoming event to which the rule is reacting.

`rule`: Object of class `Rule`, the rule being tested.

`queryResult`: Object of class `"logical"` which specifies whether or not the rule `Conditions` are met.

`final`: Object of class `Status` which describes the expected final state of the system after applying the rule, or an object of class `P4Message` describing the generated message.

Extends

Class `"EITest"`, directly. This difference is two extra slots: One for

Methods

as.jlist `signature(obj = "RuleTest", ml = "list")`: helper function for converting the test into a JSON object, see `as.json`.

queryResult `signature(x = "RuleTest")`: Returns the expected value of the `Conditions` part of the rule.

rule `signature(x = "RuleTest")`: Returns the rule being tested.

toString `signature(x = "RuleTest")`: Returns a string describing the rule.

Author(s)

Russell G. Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

See Also

The functions `testQuery`, `testPredicate`, and `testRule` are used to actually execute the tests.

The functions `RuleTest` and `parseRuleTest` are used to construct the rule.

The functions `name`, `doc`, `initial`, `event`, `rule`, `queryResult`, and `final` access the components of the test.

The `TestSet` maintains a collection of tests for a particular application.

Examples

```
showClass("RuleTest")
```

---

ruleType *Accessors for Rule objects.*

Description

The functions described here access the corresponding fields of an `Rule` object.

Usage

```
ruleType(x)
## S4 method for signature 'Rule'
ruleType(x)
priority(x)
## S4 method for signature 'Rule'
priority(x)
condition(x)
```

```
## S4 method for signature 'Rule'
condition(x)
predicate(x)
## S4 method for signature 'Rule'
predicate(x)
```

Arguments

x	An object of class Rule to be accessed.

Value

The function ruleType returns a string which should be one of "Status", "Observable", "Context", "Trigger", or "Reset".

The function priority returns a numeric value with lower values indicating higher priority.

The functions condition and predicate return a list which represents either the Conditions or Predicates of the rule.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. https://docs.mongodb.com/manual/.

See Also

The Rule object has documentation about the type and priority system and Conditions and Predicates each have detailed descriptions of the condition and predicate arguments.

Other classes in the EIEvent system: EIEngine, Context, Status, Event, RuleTable.

Other methods for working with Rules: Rule, parseRule, verb, object, context, name, doc

Examples

```
r1 <- Rule(name="Coin Rule",
          doc="Set the value of the badge to the coin the player earned.",
          app="ecd://coe.fsu.edu/PPtest",
              verb="satisfied", object="game level",
              context="ALL",
              ruleType="Observable", priority=5,
              condition=list("event.data.badge"=c("silver","gold")),
              predicate=list("!set"=c("state.observables.badge"=
                                    "event.data.badge")))

stopifnot(ruleType(r1)=="Observable", priority(r1)==5,
  all.equal(condition(r1),list("event.data.badge"=c("silver","gold"))),
  all.equal(predicate(r1),list("!set"=c("state.observables.badge"=
                                    "event.data.badge"))))
```

---

runRule — *Runs a specific rule in a particular application.*

Description

The function `runRule` runs a `Rule` against a particular `Status` (state) and `Event`. The function `runTRule` is a special version for trigger rules that sends the generated message, if any, to the listeners.

Usage

```
runRule(state, event, rule, phase)
runTRule(state, event, rule, listeners)
```

Arguments

`state`	An object of class `Status` which describes the current state for the person.
`event`	An object of class `Event` which describes the current event being processed.
`rule`	An object of class `Rule` which is the rule to be executed.
`phase`	An object of type character giving the phase being executed, used mainly for logging and error reporting.
`listeners`	An object of class `Listener` (often a `ListenerSet` which will receive the messages from the trigger rules.

Details

The function `runRule()` runs the rule logic. For more about rule logic, see `Rule`.

The function first runs `checkCondition` to check if the rule is statisfied or not. If the condition is satisfied, then the predicate of the rule will be run using `executePredicate`. The resulting state object is returned.

The function `runTRule()` is similar, but meant specifically for trigger rules. Instead of `executePredicate`, the function `buildMessages` is used to build a list of messages. These are then sent to the `listenerSet` argument using its `notifyListeners` method.

The rules are executed under `withFlogging` protection. This means that if an error is encountered, the error message is logged (along with debugging information depending on the current logging threshold). In this case, and object of class `try-error` is returned instead of the normal error return.

Value

The function `runRule` returns the modified `Status` object. The function `runTRule` returns a list of `P4Message` objects. In either case, if an error occurs, then an object of class `try-error` is returned instead of the normal value.

Note

This function uses the `flog.logger` mechanism. The following information is reported at various thresholds:

TRACE The results of `checkCondition` are reported.

DEBUG When an error occurs information about the state, event and rule where the error occurred as well a stack trace are logged.

INFO and above If an error occurs the error is logged along with context information.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

See Also

Conditions and Predicates each have detailed descriptions. The functions `checkCondition` and `executePredicate` run the condition and predicate parts of the rule. The functions `runRule` and `runTRule` run the indivual rules, and the functions `runTriggerRules`, `runStatusRules`, `runObservableRules`, `runResetRules`, and `runContextRules` run sets of rules.

The functions `testQuery` and `testQueryScript` can be used to test that rule conditions function properly. The functions `testPredicate` and `testPredicateScript` can be used to test that predicates function properly. The functions `testRule` and `testRuleScript` can be used to test that rule conditions and predicates function properly together. The class `RuleTest` stores a rule test.

Other classes in the EIEvent system: `EIEngine`, `Context`, `Status`, `Event`, `RuleTable`.

Examples

```
### runRule

st0 <- Status(uid="Test0",context="Stairs",
              timestamp=as.POSIXct("2018-09-25 12:12:28 EDT"),
              observables=list(agentsUsed=list(),
                              lastAgent=NA))
evnt1 <- Event(uid="Test0",
               verb="identified",object="game object",
               context="Stairs",
               timestamp=as.POSIXct("2018-09-25 12:12:29 EDT"),
               details= list("gemeObjectType"="Lever"))
st1exp <- Status(uid="Test0",context="Stairs",
             timestamp=as.POSIXct("2018-09-25 12:12:28 EDT"),
             observables=list(agentsUsed=list("Lever"),
                              lastAgent="Lever"))
r2o <- Rule(name="Update agent used.",
            doc="Adds the agent to the agent list, and sets the last agent.",
            context="Sketching", verb="identified",
            object="game object", ruleType="Observable",
            condition=list(event.data.gemeObjectType=c("Ramp", "Lever",
             "Springboard", "Pendulum")),
            predicate=list("!set"=list("state.observables.lastAgent"=
                                     "event.data.gemeObjectType"),
                           "!push"=list("state.observables.agentsUsed"=
                                      "event.data.gemeObjectType")))

st1act <- runRule(st0,evnt1,r2o,"Observable")
stopifnot(all.equal(st1exp,st1act))

### runTRule
st2 <- Status(uid="Test0",context="Stairs",
             timestamp=as.POSIXct("2018-09-25 12:13:30 EDT"),
             observables=list(agentsUsed=list("Lever"),
                              lastAgent="Lever",
                              badge="silver"))
evnt2 <- Event(uid="Test0",
               verb="satisfied",object="game level",
               context="Stairs",
               timestamp=as.POSIXct("2018-09-25 12:13:30 EDT"),
               details= list("badge"="silver"))
r4t <- Rule(name="Satisfied Trigger",
            doc="When the level is satisifed, send the observables.",
            context="ALL", verb="satisfied",
            object="game level", ruleType="Trigger",
            condition=list(),
            predicate=list("!send"=list(mess="Observables Available",
                                      context="state.oldContext",
                                      data=list())))

mess1 <- buildMessages(predicate(r4t),st2,evnt2)[[1]]

cl <- new("CaptureListener")


runTRule(st2,evnt2,r4t,cl)
mess1a <- cl$lastMessage()
stopifnot(all.equal(mess1a,mess1,check_ids=FALSE))
```

`runStatusRules` *Runs all of the appropriate rules of the given type.*

Description

Given a player `Status` (`state`) and an `Event`, this function runs all of the rules of the appropriate type and in order of the `priority`.

Usage

```
runStatusRules(eng, state, event, rules)
runObservableRules(eng, state, event, rules)
runContextRules(eng, state, event, rules)
runTriggerRules(eng, state, event, rules)
runResetRules(eng, state, event, rules)
```

Arguments

`eng`	An object of class `EIEngine` which is responsible for maintaining the rule base and running the rules.
`state`	An object of class `Status` which gives the state of the system before the rules are run.
`event`	An object of class `Event` which provides details of the event being processed.
`rules`	An list of objects of class `Rule` sorted in priority. The rules of the appropriate type will be selected from this list.

Details

The functions are run by `handleEvent` in the following five phases:

1. `runStatusRules` runs (calls `runRule` on) the status `Rule` objects. These update the internal status (flags and timers) of the `state`. The modified state is returned from these rules.
2. `runObservableRules` runs the observable `Rule` objects. These update the external status (observables) of the `state`. The modified state is returned from these rules.
3. `runContextRules` runs the context `Rule` objects. These update the `context` field of the state. These are run until the context changes (is no longer equal to `oldContext`) or all of the rules have been processed with no change in context. The modified state is returned from these rules.
4. `runTriggerRules` runs (calls `runTRule` on) the trigger `Rule` objects. Trigger rules do not modify the state, but rather if they are satisfied sends a message to the `listeners(eng)`.
5. `runResetRules` runs (calls `runRule` on) the reset `Rule` objects, but only when the context has changed (is different from `oldContext`). It resets counters, timers and flags. Note that the logic about whether the state has changed is in `handleEvent` not this function.

The rules are loaded from the database using the `findRules` method of the `EIEngine` (which in turn calls the `findRules` method of the `RuleTable` it contains). The rule matches the current phase if the following conditions hold:

- The `app` field of the rule matches that of the engine (and the state and event).
- The `verb` and `object` of the rule either match those of the event, or have the special value "ALL".
- The `context` of the rule either matches the context of the state, or is a context group that contains the context of the state (see `applicableContexts`) or is the special value "ALL".
- The `ruleType` of the rule matches the current phase.

All rules matching these conditions are returned and sorted by the rule's `priority`. Ties are handled arbitrarily. The rules are then run (using `runRule`) in the returned order. If an error occurs in running any of the rules, then further processing will stop and an object of class `try-error` will be returned.

Two exceptions: First, the context rules are only run until the context changes. As soon as the context changes, the function `runContextRules` exits and returns the modified state. Second, the function `runTriggerRules` uses `runTRule` instead of `runRule`. Instead of modifiying the state, it sends messages to the engine's listeners.

Rules are processed in the `withFlogging` environment, so depending on the current threshold, various information will be provides about which rules are run. Also, if an error occurs, all of the functions will return an object of class `try-error` instead of their normal return value. This can be used to suspend processing.

Value

If processing was successful the possibly modified `state` (an object of class `Status` will be returned.

If processing was unsuccessful, then an object of class `try-error` will be returned.

Note

The signature of this function has changed for reasons of efficiency. In previous versions, each function did a very simlilar database check. In practice, a lot of time was spent checking the database to find that there were no applicable rules. Now, the function `processEvent` finds the applicable rules, and the functions merely select the ones that match the status. If no rules are available, a considerable amount of time is saved.

This function uses the `flog.logger` mechanism. The following information is reported at various thresholds:

TRACE
- The results of `checkCondition` are reported.
- The specific rules found from each query are reported.
- A message is logged as each rule is run.

DEBUG
- When an error occurs information about the state, event and rule where the error occurred as well a stack trace are logged.
- Rule searches are logged and the number of rules reported.
- A message is logged when each phase starts.
- A message is logged when the context changes.

INFO and above If an error occurs the error is logged along with context information.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the rule system. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. https://docs.mongodb.com/manual/.

See Also

Conditions and Predicates each have detailed descriptions. The functions `checkCondition` and `executePredicate` run the condition and predicate parts of the rule. The functions `runRule` and `runTRule` run the indivual rules, and the functions `runTriggerRules`, `runStatusRules`, `runObservableRules`, `runResetRules`, and `runContextRules` run sets of rules.

The functions `testQuery` and `testQueryScript` can be used to test that rule conditions function properly. The functions `testPredicate` and `testPredicateScript` can be used to test that predicates function properly. The functions `testRule` and `testRuleScript` can be used to test that rule conditions and predicates function properly together. The class `RuleTest` stores a rule test.

Other classes in the EIEvent system: `EIEngine`, `Context`, `Status`, `Event`, `RuleTable`.

Examples

```
## Not run:
processEvent <- function (eng,state,event) {
  flog.debug("")                        #Blank line for visibility.
  flog.info("New Event for
             verb(event),object(event),
             toString(timestamp(event)))
  rules <- eng$findRules(verb(event),object(event),context(state))
  if (length(rules)==0L) {
    flog.info("No rules for event, skipping.")
    return (NULL)
  }
  out <- runStatusRules(eng,state,event)
  if (is(out,'try-error')) return (out)
  else state <- out
  out <- runObservableRules(eng,state,event)
  if (is(out,'try-error')) return (out)
  else state <- out
  out <- runContextRules(eng,state,event)
```

```
    if (is(out,'try-error')) return (out)
    else state <- out
    runTriggerRules(eng,state,event)
    if (oldContext(state) != context(state)) {
      out <- runResetRules(eng,state,event)
      if (is(out,'try-error')) return (out)
      else state <- out
      state@oldContext <- context(state)
    }
    state@timestamp <- timestamp(event)
    state
  }


  ## End(Not run)
```

`runTest` *Runs a test case with a given set of rules*

Description

This function runs a test case in the context of a particular application. In particular, it checks to make sure that the rules are behaving as expected.

Usage

```
runTest(eng, test)
```

Arguments

`eng`	An object of class `EIEngine` which provides the testing environment.
`test`	An object of class `EITest` which describes the test.

Details

An `EITest` object contains an `initial` `Status`, an `Event` and a result field which could either be a final `Status` or a `P4Message` that could be sent by a trigger rule. The function `runTest` processes this event with the given initial status and checks the actual result agains the expected result.

Value

The function returns `TRUE` if the event was handled without error and the result matched the expected value, `FALSE` if the event was handled without error, but the result did not match, and `NA` if handling the event (or comparing the result to the expected result) produced an error.

Note that errors are logged using `withFlogging` which will provide details about errors.

Note

This function is pretty much untested. This is a planned expansion.

Author(s)

Russell Almond

See Also

See `testRule` for the testing functions that are currently implemented.

See `EITest` for the test class.

Examples

```
## The function is currently defined as
function (eng, test)
{
    cl <- new("CaptureListener")
    eng$ListenerSet$addListener(name(test), cl)
    flog.info("Running Test %s", name(test))
    flog.debug("Details:", doc(test), capture = TRUE)
    result <- NA
    withFlogging({
        actual <- eng$testRules(initial(test), event(test))
        if (is(final(test), "P4Message")) {
            actual <- cl$lastMessage()
        }
        else if (is(final(test), "list")) {
            actual <- cl@messages
        }
        result <- all.equal(final(test), actual)
        if (!isTRUE(result)) {
            flog.info("Test %s failed.", name(test))
            flog.info("Details:", result, capture = TRUE)
            flog.debug("Actual Status/Message:", actual, capture = TRUE)
            result <- FALSE
        }
    }, context = paste("Running Test", name(test)), test = test)
    eng$ListenerSet$removeListener(name(test))
    result
  }
```

`setJS` *Sets a field in a status object in Javascript notation.*

Description

Fields of a `Status` can be accessed using JavaScript notation, e.g., `state.flags.`*field*, `state.observables.`*field*,or `state.timers.`*name*. The function `getJS` set the current value of the referenced field from the state object.

Usage

```
setJS(field, state, now, value)
setJSfield(target, fieldlist, value)
```

Arguments

`field`	A character scalar describing the field to be referenced (see details).
`state`	An object of type `Status` giving the current status of the user in the system; this argument will be modified.
`now`	An object of class `POSIXt` which gives the time ofthe event. Used when setting timers.
`value`	The new value to be assigned to the field.
`target`	A collection object to be accessed. The object implmenting `state.flags`, `state.observables` or one of sub-components.
`fieldlist`	The successive field names as a vector of characters (split at the '.' and excluding the initial `state.flags`, `state.observables` or `event.details`.

Details

The Predicates of `Rule` objects update parts of the current `state`. As these rules are typically written in JSON, it is natural to reference the parts of the `Status` objects using javascript notation. Javascript, like R, is a weakly typed language, and so javascript objects are similar to R lists: a named collection of values. A period, '.', is used to separate the object from the field name, similar to the way a '$' is used to separate the field name from the object reference when working with R lists. If the object in a certain field is itself an object, a succession of dots. Thus a typical reference looks like: *object.field.subfield* and so forth as needed.

In EIEvent rules, only the `state`, the current `Status`, can be modified. Therefore, in the predicate all dot notation field references start with `state` (field references starting with `event` can be used in the `value`). Furthermore, `Status` objects have only a certain number of settable fields so only those fields can be referenced.

The state object has two fields which are collections of arbitrary object: `state.flags` and `state.observables`. (The state also contains a collection of timer objects, `state.timers`, which has special rules described below.) Each of these is a named collection (list in R), and components can be refenced by name. The expressions "`state.flags.`*name*", and "`state.observables.`*name*" reference an object named *name* in the flags or observables field of the state respectively. Note that the available components of these lists fields will depend on the context of the simulation and the verb and object of the event.

The fields `state.flags` and `state.observables` could also be multipart objects (i.e., R lists). Additional dots can be used to reference the subcomponents. Thus "`state.flags.position.x`" references the x-coordinate of the position object in the flag field. These dots can be nested to an aribrary depth.

The fields of `state.flags.` and `state.observables.` can also contain unnamed vectors (either character, numeric, or list). In this case square brackets can be used to index the elements by position. Indexes start at 1, as in R. For example, "`state.flags.agentList[3]`" references the third value in the agentList flag of the status. Currently only numeric indexes are allowed, variable references are not, nor can sublists be selected.

Note that the fields allowed to be set are a subset of the fields in which can be accessed (see `setJS` for a complete list). In particular, only fields of the `state` object can be set, while fields of the `event` object can also be referenced. Also, certain fields of the `state` object are read only.

The function `setJSfield` is an internal function which is used to set the components, it is called recursively to modify fields which are themselves lists or vectors.

Value

The `setJS` function always returns the modified state object. The `setJSfield` function returns the modified colleciton object, or if the fieldlist is empty, the new value.

Fields of the `Status` object.

The following fields of the `Status` object can be set:

- `state.context` The current context that the state object is in.
- `state.observables.`*field* The value of the observable named *field*.
- `state.timers.`*field* The the timer named *field*. Note that `state.timers.`*field*`.time` or `.value` refers to the current elapsed time of the timer, and `state.timers.`*field*`.run` or `.running` is a logical value which refers to whether or not the timer is running.
- `state.flags.`*field* The value of the observable named *field*.

The other fields listed in `getJS` can be accessed but not set.

Timers

The `state.timers` field holds a named list of objects of class `Timer`. These behave as if they have two settable subfields: `running` (or `run`) and `time` (or `value`).

The `running` (or `run`) virtual field is a logical field: `TRUE` indicates running and `FALSE` indicates paused. Setting the value of the field will cause the timer to resume (start) or pause depending on the value.

The `time` (or `value`) field gives the elapsed time of the timer. Setting the field to zero will reset the timer to zero, setting it to another value will adjust the time.

Note

It is clear that some kind of indirect reference (i.e., using variables, either integer or character, inside of the square brackects) is needed. This may be implemented in a future version.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the rule system. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. https://docs.mongodb.com/manual/.

See Also

The functions getJS for accessing fields and removeJS for removing fields (only allowed with state objects). This function is called from executePredicate.

The help files Conditions and Predicates each have detailed descriptions of rule syntax.

Other classes in the EIEvent system: EIEngine, Context, Status, Event, Rule.

Examples

```
st <- Status("Phred","Level 0",timerNames="watch",
   flags=list("list"=list("one"=1,"two"="too"),"vector"=(1:3)*10,
              "char"="hello"),
   observables=list("list"=list("one"="one","two"=2),"vector"=(1:3),
           "char"="bar"),
   timestamp=as.POSIXct("2018-12-21 00:01"))

ev <- Event("Phred","test","message",
      timestamp=as.POSIXct("2018-12-21 00:01:01"),
      details=list("list"=list("one"=1,"two"=1:2),"vector"=(1:3)))

ts <-timestamp(ev)

st <- setJS("state.context",st,ts,"Level 1")
stopifnot(getJS("state.context",st,ev)=="Level 1",
          getJS("state.oldContext",st,ev)=="Level 0")

st <- setJS("state.observables.numeric",st,ts,12.5)
stopifnot( getJS("state.observables.numeric",st,ev)==12.5)

st <- setJS("state.observables.char",st,ts,"foo")
stopifnot(getJS("state.observables.char",st,ev)=="foo")

st <- setJS("state.observables.list.one",st,ts,"a")
stopifnot(
  all.equal(getJS("state.observables.list",st,ev),list("one"="a","two"=2)),
  getJS("state.observables.list.one",st,ev)=="a")
```

```
st <- setJS("state.observables.vector",st,ts,(1:3)*100)
st <- setJS("state.observables.vector[2]",st,ts,20)
stopifnot(
  getJS("state.observables.vector[2]",st,ev)==20,
  all.equal(getJS("state.observables.vector",st,ev),c(100,20,300)))


st <-setJS("state.flags.list.two",st,ts,"two")
stopifnot(all.equal(getJS("state.flags.list",st,ev),list("one"=1,"two"="two")))

st <-setJS("state.flags.vector[3]",st,ts,3)
stopifnot(
  getJS("state.flags.vector[3]",st,ev)==3,
  all.equal(getJS("state.flags.vector",st,ev),c(10,20,3)))

st <- setJS("state.flags.logical",st,ts,TRUE)
stopifnot( getJS("state.flags.logical",st,ev)==TRUE)

st <- setJS("state.flags.char",st,ts,"foobar")
stopifnot(getJS("state.flags.char",st,ev)=="foobar")
```

`setTimer` *Manipulates a Timer inside of a Status*

Description

A `Status` object contains a named collection of `Timer` objects. These functions access the timer object. Note that because the timer is not an actual clock, but rather calculates the time difference between events, most of the functions must be passed the “current” time, which is usually the `timestamp` of the `Event` object being processed.

Usage

```
setTimer(x, timerID, time, running, now)
## S4 method for signature 'Status,character'
setTimer(x, timerID, time, running, now)
timer(x, name)
timer(x, name) <- value
timerTime(x, name, now)
## S4 method for signature 'Status,character'
timerTime(x, name, now)
timerTime(x, name, now) <- value
## S4 replacement method for signature 'Status,character'
timerTime(x, name, now) <- value
## S4 method for signature 'Status,character'
timerRunning(x, name, now)
timerRunning(x, name, now) <- value
```

```
## S4 replacement method for signature 'Status,character'
timerRunning(x, name, now) <- value
```

Arguments

`x`	An object of class `Status` whose timers are to be accessed. (These are generic functions, so methods for classes other than `Status` could be written.)
`timerID`	A character string of the form `state.timers.`*name*. The name operates like the name argument.
`name`	A character scalar giving the name of the timer to be accessed.
`time`	The new elapsed time of the timer. This should be an object of class `difftime`.
`running`	A logical flag indicating whether or not the timer should be running.
`now`	The current time, usually from the `timestamp` of the `Event` object being processed.
`value`	The replacement value. For `timerTime<-` this should be an object of class `difftime`. For `timerRunning<-` this should be a logical value. For `timer<-` this should be an object of class `Timer`.

Details

The `Status` objects contain a named list of `Timer` objects. Each timer contains two conceptual fields: `running` which indicates whether or not the timer is running and `time` which indicates the current elapsed time. (Note that these are actually implemented using differences between timestamps, which is why most of the functions need to pass the current time in the `now` argument. See `start` for details.

The `timer` and `timer<-` functions access the `Timer` object directly.

The `timerRunning` and `timerRunning<-` functions access the conceptual `running` field of the timer. In particular, the setter method `start`s or `pause`s the timer.

The `timerTime` and `timerTime<-` functions access the conceptual `time` field of the timer. In particular, they call `timerTime` or `timerTime` on the timer.

The function `setTimer` is an omnibus modifier, meant to be called from `setJS`. Instead of the timer name, it uses the fully qualified dot notation: `state.timers.`*name*. If no timer for the given name exists, it creates one; otherwise, it uses the existing timer. It then calls `timerTime<-` and `timerRunning<-` with the given value.

The functions `getJS` and `setJS` call these functions if the referenced field is of the form `state.timers.`*name*. Note that the `setJS` function gets the current time from the event object, so it does not need to be specified in this form.

Value

The function `timer` returns an object of class `Timer`.

The function `timerRunning` returns a logical value indicating whether or not the timer is currently running.

The function `timerTime` returns the elapsed time in `difftime` format.

The function `setTimer` and the other setter methods return the `Status` object which is the first argument.

Note

Internally, the timers are implemented as a start time and an elapsed time (see `start`). Elapsed times are calculated by differencing two time stamps. Therefore, it is usually necessary to pass along the “current” time with these functions, usually from the timestamp of the `Event` object.

Author(s)

Russell Almond

See Also

The `Timer` class describes timers, and the `Status` class contains a collection of timers.

Methods for manipulating timer directly include `start`, `resume`, `pause`, `isRunning`, `timeSoFar` and `reset`.

The functions `setJS`, `getJS` and `removeJS` have details about how to manipulate timers using rules.

Examples

```
st <- Status("Phred","Level 0",timerNames="watch",
   timestamp=as.POSIXct("2018-12-21 00:00:01"))
context(st) <- "Level 1"

stopifnot(timer(st,"watch")@name=="watch")

timer(st,"stopwatch") <- Timer("stopwatch")
stopifnot(timer(st,"stopwatch")@name=="stopwatch")

timerRunning(st,"stopwatch",as.POSIXct("2018-12-21 00:00:01")) <- TRUE
stopifnot(!timerRunning(st,"watch",as.POSIXct("2018-12-21 00:00:02")),
           timerRunning(st,"stopwatch",as.POSIXct("2018-12-21 00:00:02")))

timerRunning(st,"stopwatch",as.POSIXct("2018-12-21 00:01:01")) <- FALSE
stopifnot(!timerRunning(st,"stopwatch",as.POSIXct("2018-12-21 00:01:02")),
           timerTime(st,"stopwatch",as.POSIXct("2018-12-21 00:01:02"))==
           as.difftime(1,units="mins"))

timerRunning(st,"stopwatch",as.POSIXct("2018-12-21 00:03:01")) <- TRUE
stopifnot(timerTime(st,"watch",as.POSIXct("2018-12-21 00:03:02"))==
          as.difftime(0,units="secs"),
          timerTime(st,"stopwatch",as.POSIXct("2018-12-21 00:03:02"))==
          as.difftime(61,units="secs"))


timerTime(st,"watch",as.POSIXct("2018-12-21 00:05:00")) <-
         as.difftime(5,units="mins")
timerTime(st,"stopwatch",as.POSIXct("2018-12-21 00:05:00")) <-
```

```
         as.difftime(5,units="mins")
stopifnot(timerTime(st,"watch",as.POSIXct("2018-12-21 00:05:02"))==
          as.difftime(300,units="secs"),
          timerTime(st,"stopwatch",as.POSIXct("2018-12-21 00:05:02"))==
          as.difftime(302,units="secs"))


st <- setTimer(st,"state.timers.stopwatch",as.difftime(0,units="secs"),FALSE,
               as.POSIXct("2018-12-21 00:10:00"))
st <- setTimer(st,"state.timers.runwatch",as.difftime(50,units="secs"),TRUE,
               as.POSIXct("2018-12-21 00:10:00"))
stopifnot(!timerRunning(st,"stopwatch",as.POSIXct("2018-12-21 00:10:02")),
           timerRunning(st,"runwatch",as.POSIXct("2018-12-21 00:10:02")),
           timerTime(st,"stopwatch",as.POSIXct("2018-12-21 00:10:02"))==
           as.difftime(0,units="secs"),
           timerTime(st,"runwatch",as.POSIXct("2018-12-21 00:10:02"))==
           as.difftime(52,units="secs"))
```

start *Functions for manipulating timer objects.*

Description

A Timer object keeps track of the elapsed time between events. These functions update the state of the timer. Note that because Timers don't operate in real time, most of these functions need to be passed the "current" time, which is the timestamp of the Event which is being processed.

Usage

```
start(timer, time, runningCheck = TRUE)
## S4 method for signature 'Timer,POSIXt'
start(timer, time, runningCheck = TRUE)
pause(timer, time, runningCheck = TRUE)
## S4 method for signature 'Timer,POSIXt'
pause(timer, time, runningCheck = TRUE)
resume(timer, time)
## S4 method for signature 'Timer,POSIXt'
resume(timer, time)
isRunning(timer)
## S4 method for signature 'Timer'
isRunning(timer)
timeSoFar(timer, time)
## S4 method for signature 'Timer,POSIXt'
timeSoFar(timer, time)
## S4 replacement method for signature 'Timer,POSIXt'
timeSoFar(timer, time) <- value
totalTime(timer)
```

```
## S4 method for signature 'Timer'
totalTime(timer)
reset(timer)
## S4 method for signature 'Timer'
reset(timer)
```

Arguments

`timer`	An object of class `Timer`.
`time`	Either an object of class `POSIXt` or a an object that can be coerced into class `POSIXt`. Note that strings need to be in ISO 8601 format (or manually converted using `strptime`.) This is usually the `timestamp` value of the `Event` being processed.
`runningCheck`	A logical value. If `TRUE` `start`, `pause` and `resume` will signal an error if the timer is currently in an unexpected state.
`value`	A time interval in `difftime` format which is to be the new elapsed time.

Details

These functions allow the `Timer` object to behave like a stopwatch, event though it is implemented with a collection of timestamps. Because it doesn't really run a clock, but instead takes the different between the timestamps of the starting and finishing event, most of the function here need to pass the "current" time, which is defined as the `timestamp` of the `Event` object being processed.

The following functions are supported:

start Sets the timer running. If `runningCheck` is true, it signals an error if the timer is already running. Does not change the elapsed time.

pause Pauses the timer. If `runningCheck` is true, it signals an error if the timer is not currently running.

resume Identical to start with `runningCheck=TRUE`.

timeSoFar Returns the elapsed time on the timer. (Similar to the lap time on a stopwatch.)

timeSoFar<- Sets the elapsed time to a new value. If the timer is running, adjusts the start time to the current `time` argument.

isRunning Returns the state of the timer as a logical value (`TRUE` for running).

reset Stops the timer and sets the elapsed time to zero.

totalTime Returns the total time up until the last pause; if the timer is running, the time since `startTime` is ignored.

Value

The `isRunning` function returns a logical value giving the state of the timer.

The `timeSoFar` and `totalTime` functions returns an object of type `difftime` giving the current elapsed time.

The other functions return the modified `timer` object.

Note

Internally, `Timer` objects maintain their state using two fields: `startTime` and `totalTime`. When the timer is started (resumed), the `startTime` is set to the current time, and it is set to `NA` when it is paused. The running status can be determined by checking whether or not the `startTime` is NA. When the timer is paused, the difference between the current time and the `startTime` is added to the `totalTime`. So the elapsed time is the `totalTime` plus the difference between the current `time` (the argument) and the `startTime`.

Here are the acutal implementation of the manipulation functions:

start Sets the `startTime` to the `time`. Does not change `totalTime`.

pause Adds the difference between `time` and `startTime` to `totalTime`; sets `startTime` to `NA`.

resume Identical to start with `runningCheck=TRUE`.

timeSoFar If running, returns `totalTime` + `time` - `startTime`. If paused, returns `totalTime`.

timeSoFar<- Sets `totalTime` to `value`. If the timer is running sets `startTime <- time`.

isRunning Returns `!is.na(startTime)`.

reset Sets `startTime` to `NA` and `totalTime` to 0 seconds.

totalTime Returns the value of the `totalTime` field, ignoring the time elapsed between `time` and `startTime`.

Author(s)

Russell Almond

See Also

The `Timer` class describes timers, and the `Status` class contains a collection of timers.

Methods for manipulating timers in states: `timer`, `timerTime`, and `timerRunning`

The functions `setJS`, `getJS` and `removeJS` have details about how to manipulate timers using rules.

Examples

```
## Create the timer
stopwatch <- Timer("stopwatch")
stopifnot(isRunning(stopwatch)==FALSE,
          timeSoFar(stopwatch,as.POSIXct("2018-12-21 00:01"))==
          as.difftime(0,units="mins"),
          totalTime(stopwatch)==as.difftime(0,units="mins"))

## Start the timer.
stopwatch <- start(stopwatch,as.POSIXct("2018-12-21 00:01"))
stopifnot(isRunning(stopwatch)==TRUE,
          timeSoFar(stopwatch,as.POSIXct("2018-12-21 00:01"))==
          as.difftime(0,units="mins"),
          totalTime(stopwatch)==as.difftime(0,units="mins"))
```

```
## Note that time so far is based on the time argument.
stopifnot(isRunning(stopwatch)==TRUE,
          timeSoFar(stopwatch,as.POSIXct("2018-12-21 00:10"))==
          as.difftime(9,units="mins"),
          totalTime(stopwatch)==as.difftime(0,units="mins"))

## Pause the timer.
stopwatch <- pause(stopwatch,as.POSIXct("2018-12-21 00:10"))
stopifnot(isRunning(stopwatch)==FALSE,
          timeSoFar(stopwatch,as.POSIXct("2018-12-21 00:10"))==
          as.difftime(9,units="mins"),
          totalTime(stopwatch)==as.difftime(9,units="mins"))

## adjust the time.
timeSoFar(stopwatch,as.POSIXct("2018-12-21 00:10")) <-
    as.difftime(10,units="mins")
stopifnot(isRunning(stopwatch)==FALSE,
          timeSoFar(stopwatch,as.POSIXct("2018-12-21 00:10"))==
          as.difftime(10,units="mins"),
          totalTime(stopwatch)==as.difftime(10,units="mins"))

## resume the timer and adjust the time again.
stopwatch <- resume(stopwatch,as.POSIXct("2018-12-21 01:01"))
timeSoFar(stopwatch,as.POSIXct("2018-12-21 01:10")) <-
    as.difftime(5,units="mins")
stopifnot(isRunning(stopwatch)==TRUE,
          timeSoFar(stopwatch,as.POSIXct("2018-12-21 01:11"))==
          as.difftime(6,units="mins"),
          totalTime(stopwatch)==as.difftime(5,units="mins"))

## Reset the timer
stopwatch <- reset(stopwatch)
stopifnot(isRunning(stopwatch)==FALSE,
          timeSoFar(stopwatch,as.POSIXct("2018-12-21 00:01"))==
          as.difftime(0,units="mins"),
          totalTime(stopwatch)==as.difftime(0,units="mins"))
```

`Status` *Status (state) object constructor*

Description

The `Status` function is the constructor for the `Status` (or state) object. As Status objects are usually read from a database or other input stream, the `parseStatus` function is recreates an event from a JSON list and `as.jlist` encodes them into a list to be saved as JSON.

Usage

```
Status(uid, context, timerNames = character(), flags = list(), observables = list(), timestamp = Sys.ti
parseStatus(rec)
## S4 method for signature 'Status,list'
as.jlist(obj, ml, serialize = TRUE)
```

Arguments

`uid`	A character scalar identifying the examinee or player.
`context`	A character string describing the task, item or game which the player is currently in.
`timerNames`	A list of names for timer objects. These are created calling `Timer` with each name as an argument.
`flags`	A named list containing internal details about the event. The necessary fields will depend on the `app` and the `context`.
`observables`	A named list containing external details about the event. The necessary fields will depend on the `app` and the `context`.
`timestamp`	The timestamp of the most recent `Event` processed.
`app`	A character scalar providing a unique identifier for the application (game or assessment). This defines the available vocabulary for `flags` `timers` and `observables`, as well as the set of applicable `Rule` objects.
`rec`	A named list containing JSON data.
`obj`	An object of class `Status` to be encoded.
`ml`	A list of fields of `obj`. Usually, this is created by using `attributes(obj)`.
`serialize`	A logical flag. If true, `serializeJSON` is used to protect the `data` field (and other objects which might contain complex R code.

Details

Most of the details about the `Status` object, and how it works is documented under Status-class. Note that the distinction between `flags` and `observables` is mostly one of intended usage: `observables` are reproted to other processes, and `flags` are not. Also, `timers` are created by their name and then need to be specifically set.

The function `as.jlist` converts the `obj` into a named list. It is usually called from the function `as.json`.

The `parseStatus` function is the inverse of `as.jlist` applied to a status object. It is designed to be given as an argument to `getOneRec` and `getManyRecs`.

Value

The functions `Status` and `parseStatus` return objects of class status. The function `as.jlist` produces a named list suitable for passing to `toJSON`.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the JSON layout of the Status/State objects. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

Betts, B, and Smith, R. (2018). The Leraning Technology Manager’s Guid to xAPI, Second Edition. HT2Labs Research Report: https://www.ht2labs.com/resources/the-learning-technology-managers-guide-to- #gf_26.

HT2Labs (2018). Learning Locker Documentation. https://docs.learninglocker.net/welcome/.

See Also

`Status` describes the state object.

`parseMessage` and `as.json` describe the JSON conversion system.

The functions `getOneRec` and `getManyRecs` use `parseStatus` to extract events from a database.

Examples

```
st <- Status("Phred","Level 0",timerNames=c("watch","stopwatch"),
   flags=list("list"=list("one"=1,"two"="too"),"vector"=(1:3)*10),
   observables=list("numeric"=12.5,char="foo",
                 "list"=list("one"="a","two"=2),"vector"=(1:3)*100),
   timestamp=as.POSIXct("2018-12-21 00:01"))

st <- setTimer(st,"state.timers.stopwatch",as.difftime(15,units="secs"),TRUE,
               as.POSIXct("2018-12-21 00:01"))

sta <- parseStatus(as.jlist(st,attributes(st)))
stopifnot(all.equal(st,sta))
```

`Status-class` *Class* "Status"

Description

A `Status` object represents the state of a student (user) in the simulation. In particular, it provides lists of `flags`, `timers` and `observables` whose values are updated by the `Rule` objects after an `Event`.

Objects from the Class

Objects can be created by calls of the function `Status(...)`. Generally, the `EIEngine` maintains one status object for every student in the system.

Slots

`_id`**:** Internal database ID.

`app`**:** Object of class `"character"` giving an identifier for the application.

`uid`**:** Object of class `"character"` giving an identifier for the user or student.

`context`**:** Object of class `"character"` giving an identifier for the scoring context the student is currently in.

`oldContext`**:** Object of class `"character"` giving an identifier for the previous scoring context. In particluar, if this value is not equal to `context` it means the context has recently chagned.

`timers`**:** A named `"list"` of `Timer` objects representing events that need to be timed.

`flags`**:** A named `"list"` of fields representing the state of the system. Unlike observables, flags are generally not reported outside the system.

`observables`**:** A named `"list"` of fields representing details of the task interaction which will be reported outside of the evidence identification system.

`timestamp`**:** Object of class `"POSIXt"` giving the timestamp of the last `Event` processed for this student.

Methods

as.jlist `signature(obj = "Status", ml = "list")`: preprocessing method used to conver the `Status` into a JSON string. See `as.jlist`.

app `signature(x = "Status")`: returns the value of the `context` field.

context `signature(x = "Status")`: returns the value of the `context` field.

flag `signature(x = "Status")`: returns the list of flags associated with the status.

flag<- `signature(x = "Status")`: sets the list of flags associated with the status.

obs `signature(x = "Status")`: returns the list of observables associated with the status.

obs<- `signature(x = "Status")`: sets the list of observables associated with the status.

setTimer `signature(x = "Status", name = "character")`: sets the state of the named timer.

timer `signature(x = "Status", name = "character")`: returns the named timer object.

timer<- `signature(x = "Status", name = "character")`: sets the named timer object.

timerRunning `signature(x = "Status", name = "character")`: returns the running status of the named timer.

timerRunning<- `signature(x = "Status", name = "character")`: sets the running status of the named timer.

timerTime `signature(x = "Status", name = "character")`: returns the elapsed time of the named timer.

timerTime<- `signature(x = "Status", name = "character")`: sets the elapsed time of the named timer.

all.equal.Status `(target, current, ..., checkTimestamp=FALSE, check_ids=TRUE)`: (S3 method) Checks for equality. The `checkTimestamp` flag controls whether or not the timestamp is checked. The `check_ids` flag controls whether or not the database IDs are checked.

Note that the `getJS` and `setJS` functions are often used to get and set the values of the status object.

Header Fields

A `Status` object always has header fields. The `app` field references the application (assessment) that this state belongs to. The `uid` field is the student (user or player) the status represents. The `timestamp` field is set to the timestamp of the last event processed (after it is processed).

The `context` and `oldContext` fields operate as a pair. Before the event is processed, `oldContext` is set to the current value of `context`. If the value of `context` changes (in particular, as result of a context rule, see `Rule`), then this can be determined by comparing the value of `context` and `oldContext`.

Flags and Observables

The `flags` and `observables` fields are both named collections of arbitrary R objects. The exact values stored here will depend on the logic of the application. In general, these can be scalar numeric, character or logical variables, vectors of the same, or more complex objects made from named lists. It is probably not good to use formal S3 or S4 objects as these won't be properly saved and restored from a database.

The difference between the `flags` and `observables` is a convention that is not enforced in the code. The `observables` are intended to be reported using the trigger rules (see `Rule`). The `flags` are meant to hold intermediate values that are used to calculate `observables`. Nominally, status rules are used to update flags and observable rules to update observables, but this is not enforced.

Timers

The `timers` field holds a named list of objects of class `Timer`. These behave as if they have two settable subfields: `running` (or `run`) and `time` (or `value`).

The `running` virtual field is a logical field: `TRUE` indicates running and `FALSE` indicates paused. Setting the value of the field will cause the timer to resume (start) or pause depending on the value.

The `time` field gives the elapsed time of the timer. Setting the field to zero will reset the timer to zero, setting it to another value will adjust the time.

Dot (Javascript) Field Reference

Fields in the state object can be referenced using a pseudo-Javascript dot notation, where nested components are referenced through the '.' operator (which operates similarly to the R '$' operator). These all start with `state.` to distringuish them from fields in the event. In particular, the follwing fields are recognized.

- `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. (This field should not be set by user code).

- `state.observables.`*field* The value of the observable named *field*.
- `state.timers.`*field* The the timer named *field*. Note that `state.timers.`*field*`.time` or `.value` refers to the current elapsed time of the timer, and `state.timers.`*field*`.run` or `.running` is a logical value which refers to whether or not the timer is running.
- `state.flags.`*field* The value of the observable named *field*.
- `state.timestamp` The time at which the last processed event occurred (this field is read-only).

  The functions `getJS` and `setJS` are used to access the fields, and the help for those functions contains a number of examples.

The document "Rules Of Evidence" gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, `http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671`.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: `https://education.umd.edu/file/11333/download?token=kmOIVIwi`, Video: `https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4`.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. `https://docs.mongodb.com/manual/`.

Author(s)

Russell Almond

See Also

Other classes in the EIEvent system: `EIEngine`, `Context`, `Rule`, `Event`, `RuleTable`, `Timer`.

The functions `setJS`, `getJS` and `removeJS` have details about using the dot notation to reference fields in the status. Rule Conditions and Predicates also reference fields in the status object.

Methods for working with States: `timer`, `timerTime`, `timerRunning`, `flag`, `obs`, `app`, `context`,`oldContext`, `timestamp`, `parseStatus`,

Examples

```
showClass("Status")
```

`testRule` *Functions for testing rule queries.*

Description

The `Rule` objects in an `EIEngine` form a program, which requires testing. These functions provide a mechanism for testing the rules. The script gives a `Status`, `Event` and `Rule` object, and then checks to see if the the rule achieves the expected result or not. The functions `queryTest`, `predicateTest`, and `ruleTest` test a single rule, and the functions `queryTestScript`, `predicateTestScript`, and `ruleTestScript` test a collection of rules found in a JSON file.

Usage

```
testQuery(test)
testQueryScript(filename, suiteName = basename(filename))
testPredicate(test)
testPredicateScript(filename, suiteName = basename(filename))
testRule(test, contextSet=NULL)
testRuleScript(filename, suiteName = basename(filename), contextSet=NULL)
```

Arguments

`test`	An object of class `RuleTest`. See details.
`filename`	A pathname or URL giving a JSON file filled with rule tests.
`suiteName`	A name associated with the test scripts for reporting.
`contextSet`	A collection of contexts, used to resovle context matching issues. This should be an object which is suitable as an argument to `matchContext`, either a `list` or an object of class `ContextSet`. If `contextSet` is null, the context matching is not tested.

Details

A `test` is a `RuleTest` class which has the following components:

name An identifier for the test; used in reporting.

doc Human readable documentation; reported only if `verbose` is `TRUE`.

inital An object of class `Status` giving the initial state of the system.

event An object of class `Event` giving the current event.

rule The `Rule` object to be tested.

queryResult A logical value indicating whether or not the `Conditions` of the rule are satisfied. If this value is false, then `testPredicate` will skip the test.

final This should be an object of class `Status`. If `queryResult` is true, this should be the final state of the system after the predicate is run. If `queryResult` is false, this should be the same as the initial state.

The function `testQuery` runs `checkCondition` with arguments `(condition(rule), initial, event)` and checks the value against `queryResult`.

The function `predicateTest` runs `executePredicate` with arguments `(predicate(rule), initial, event)` and checks the value against `final`. If `queryResult` is false, the test does not run `executePredicate` and always returns true.

The function `ruleTest` does the complete testing of the rules. It checks to make sure that the `verb` and `object` of the rule and event match, and if `contextSet` is not null, it checks the context as well. It then runs first `checkCondition` and then `executePredicate` if the condition returns true. The result is checked against `final`.

The functions `testQueryScript`, `testPredicateScript` and `testRuleScript` run a suite of tests stored in a JSON file and return a logical vector of results, with an `NA` if an error was generated in the condition check or predicate execution.

Value

The functions `testQuery`, `testPredicate`, and `testRule` return a logical value indicating whether the actual result matched the expected result. If an error is encountered while processing the rule, it is caught and logged and `NA` is returned from the function.

The functions `testQueryScript`, `testPredicateScript`, and `testRuleScript` returns a logical vector indicating the result of each test. The values will be true if the test passed, false if it failed and code `NA` if either an error occured in either parsing or executing the test.

Logging

The results are logged using `flog.logger` to a logger named “EIEvent” (that is the package name). This can be redirected to a file or used to control the level of detail in the logging. In particular, `flog.appender(appender.file("/var/log/Proc4/EIEvent_log.json"), name="EIEvent")` would log status messages to the named file. Furthermore, `flog.layout(layout.json,name="EIEvent")` will cause the log to be in JSON format; useful as the inputs are logged as JSON objects which facilitates later debugging.

The amount of information can be controlled by using the `flog.threshold` command. In particular, `flog.threshold(level,name="EIEvent")`. The following information is provided at each of these levels (this is cumulative):

FATAL Fatal errors are logged.

ERROR All errors are logged.

WARN Warnings are logged.

INFO Tests are logged as they are run, as are results. Final suite results are logged.

DEBUG Test doc strings are printed. Additional context information (rule, initial, event and final) are printed on test failure. On errors, additional context information (rule, intial and event) are provided along with a stack trace.

TRACE No additional information is included at this level.

Note

The functions `testQuery`, `testPredicate`, and `testRule` suppress errors (using `withFlogging`) on the grounds that it is usually better to attempt all of the tests rather than stop at the first failure. This is also true of `testQueryScript`, `testPredicateScript`, and `testRuleScript`, which also continue after syntax errors in the test file. Certain errors, however, are not caught including errors opening the target file and the initial JSON parsing.

Author(s)

Russell Almond

References

The document “Rules Of Evidence” gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

MongoDB, Inc. (2018). *The MongoDB 4.0 Manual*. https://docs.mongodb.com/manual/.

See Also

Rule describes the rule object, Conditions describes the conditions and Predicates describes the predicates. The function checkCondition tests when conditions are satisfied, and executePredicate executes the predicate. RuleTest describes the test object.

Other classes in the EIEvent system: EIEngine, Context, Status, Event, RuleTable.

Examples

```
## Query Tests
test <- RuleTest(
  name="Simple test",
  doc="Demonstrate test mechanism.",
  initial = Status("Fred","test",timestamp=as.POSIXct("2018-12-21 00:01:01")),
  event= Event("Fred","test","rule",details=list(trophy="gold"),
         timestamp=as.POSIXct("2018-12-21 00:01:01")),
  rule=Rule(condition=list("event.data.trophy"="gold"),
            predicate=list("!set"=c("state.observables.trophy"="event.data.trophy")),
            ruleType="Observable"),
  queryResult=TRUE,
  final = Status("Fred","test",
     observables=list("trophy"="gold"),
     timestamp=as.POSIXct("2018-12-21 00:01:01")),
  )
## This test should succeed.
stopifnot(testQuery(test))

test1 <- RuleTest(
  name="Simple test",
  doc="Demonstrate test mechanism.",
  initial = Status("Fred","test",timestamp=as.POSIXct("2018-12-21 00:01:01")),
  event= Event("Fred","test","rule",details=list(trophy="silver"),
         timestamp=as.POSIXct("2018-12-21 00:01:01")),
  rule=Rule(condition=list("event.data.trophy"="gold"),
            predicate=list("!set"=c("state.observables.trophy"="event.data.trophy")),
            ruleType="Observable"),
  queryResult=TRUE,
  final = Status("Fred","test",
     observables=list("trophy"="silver"),
     timestamp=as.POSIXct("2018-12-21 00:01:01")),
  )
```

```
## This test should fail query check, query needs to allow gold or
## silver trophies.
stopifnot(!testQuery(test1))


stopifnot(all(
testQueryScript(file.path(library(help="EIEvent")$path,"testScripts",
                "CondCheck.json"))
))

## Predicate Tests
testr <- RuleTest(
  name="Simple set",
  doc="Demonstrate predicate test mechanism.",
  initial = Status("Fred","test",
    timestamp=as.POSIXct("2018-12-21 00:01:01")),
  event= Event("Fred","test","rule",details=list(agent="ramp"),
    timestamp=as.POSIXct("2018-12-21 00:01:01")),
  rule=Rule(predicate=list("!set"=c("state.observables.rampused"=TRUE)),
         ruleType="Observable"),
  queryResult=TRUE,
  final=Status("Fred","test",observables=list("rampused"=TRUE),
    timestamp=as.POSIXct("2018-12-21 00:01:01")))

stopifnot(testPredicate(testr))

testr1 <- RuleTest(
  name="Simple test",
  doc="Demonstrate test mechanism.",
  initial = Status("Fred","test",
        timestamp=as.POSIXct("2018-12-21 00:01:01")),
  event= Event("Fred","test","rule",details=list(agent="ramp"),
       timestamp=as.POSIXct("2018-12-21 00:01:01")),
  rule=Rule(predicate=list("!set"=c("state.observables.grampused"=TRUE)),
         ruleType="Observable"),
  queryResult=TRUE,
  final=Status("Fred","test",observables=list("rampused"=TRUE),
    timestamp=as.POSIXct("2018-12-21 00:01:01")))

stopifnot(!testPredicate(testr1))

stopifnot(all(
testPredicateScript(file.path(library(help="EIEvent")$path,"testScripts",
                "PredCheck.json"))
))


nocoin <- Status(uid="Test0", context="Level 84",
                 timestamp=as.POSIXlt("2018-09-25 12:12:28 EDT"),
                 observables=list(badge="none"))
scoin <- Status(uid="Test0", context="Level 84",
                 timestamp=as.POSIXlt("2018-09-25 12:12:28 EDT"),
                 observables=list(badge="silver"))
gcoin <- Status(uid="Test0", context="Level 84",
                 timestamp=as.POSIXlt("2018-09-25 12:12:28 EDT"),
                 observables=list(badge="gold"))

nevent <- Event(app="https://epls.coe.fsu.edu/PPTest",
            uid="Test0",verb="satisfied",
            object="game level", context="Level 84",
            timestamp=as.POSIXlt("2018-09-25 12:12:28 EDT"),
            details=list(badge="none"))
sevent <- Event(app="https://epls.coe.fsu.edu/PPTest",
            uid="Test0",verb="satisfied",
            object="game level", context="Level 84",
            timestamp=as.POSIXlt("2018-09-25 12:12:28 EDT"),
            details=list(badge="silver"))
gevent <- Event(app="https://epls.coe.fsu.edu/PPTest",
            uid="Test0",verb="satisfied",
            object="game level", context="Level 84",
            timestamp=as.POSIXlt("2018-09-25 12:12:28 EDT"),
            details=list(badge="gold"))

crule <- Rule(name= "Coin Rule",
            doc= "Set the value of the badge to the coin the player earned.",
            verb= "satisfied", object= "game level",
            context= "ALL", ruleType= "observables",
            priority= 2, condition= list(event.data.badge=c("silver","gold")),
            predicate= list("!set"=c(state.observables.badge =
                                    "event.data.badge")))

stopifnot(testRule(RuleTest(name="Gold coin test",initial=nocoin,event=gevent,
                   rule=crule,queryResult=TRUE,final=gcoin)))
stopifnot(testRule(RuleTest(name="Silver coin test",initial=nocoin,event=sevent,
                   rule=crule,queryResult=TRUE,final=scoin)))
stopifnot(testRule(RuleTest(name="No coin test",initial=nocoin,event=nevent,
                   rule=crule,queryResult=FALSE,final=nocoin)))



stopifnot(all(
testRuleScript(file.path(library(help="EIEvent")$path,"testScripts",
                "Coincheck.json"))
))
```

TestSet-class *Class* "TestSet"

Description

This is a database stored collection of `RuleTest` objects. It can be used to store test suites with a rule set. This is a stub implementation to be completed later.

Extends

All reference classes extend and inherit methods from `"envRefClass"`.

Fields

`app`**:** Object of class `character` giving the identifier for the application. This is part of the database key for the collection.

`dbname`**:** Object of class `character` giving the name of the database (e.g., "EIRecords").

`dburi`**:** Object of class `character` giving the uri for the database, (e.g., "mongodb://localhost").

`contexts`**:** Object of class `ContextSet` which gives the context information.

`rules`**:** Object of class `RuleTable` which gives the rules against which this test set should be run.

`db`**:** Object of class `MongoDB` which is the handle to the database collection. As this field is initialized when first requested, it should not be accessed directly, but instead through the `recorddb()` method.

Methods

`testdb()`**:** This returns the handle to the `mongo` collection object. If the connection has not yet been initialized.

`initialize(app, dbname, dburi, contexts, rules, db, ...)`**:** This sets up the object. Note that the db field is not initialized until `testdb()` is first called.

`clearAll()`**:** Clears all records from the test collection.

Note

Not yet fully implemented, this documentation is subject to change in the next version.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the rule system. `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

See Also

`runTest`, `EIEngine`, `Rule`, `testRule`

Examples

```
showClass("TestSet")
```

Timer *Constructor for Timer objects.*

Description

The `Timer` funciton is the constructor for the `Timer` object. Timer objects are usually part of a `Status` object. The `parseTimer` function recreates a timer from a JSON list and is used as part of `parseStatus`.

Usage

```
Timer(name)
parseTimer(rec)
## S4 method for signature 'Timer,list'
as.jlist(obj, ml, serialize = TRUE)
```

Arguments

`name`	A character scalar giving the name of the timer. This is used for documentation only, but it is most useful if it matches the name it is given in the `Status` object.
`rec`	A named list containing JSON data.
`obj`	An object of class `Event` to be encoded.
`ml`	A list of fields of `obj`. Usually, this is created by using `attributes(obj)`.
`serialize`	A logical flag. If true, `serializeJSON` is used to protect the `data` field (and other objects which might contain complex R code.

Details

The `Timer` function creates a new timer with zero elapsed time and in the paused (not running) state. Normally, this is not called directly, but through either the `Status` constructor or the `setTimer` function.

The function `as.jlist` converts the `obj` into a named list. It is usually called from the `as.jlist` function applied to a `Status` object, which is in turn usually called from `as.json`.

The `parseTimer` function is the inverse of `as.jlist` applied to atimer object. It is designed to be called by the function, `parseStatus`, which is given as an argument to `getOneRec`, `getManyRecs`

Value

The functions `Timer` and `parseTimer` return objects of class timer. The function `as.jlist` produces a named list suitable for passing to `toJSON`.

Note

See `start` for information about how the timer is actually implemented.

Author(s)

Russell Almond

See Also

`Timer` describes the timer object, and `Status` describes the status object which contains it.

Methods for mainpulating timers: `start`, `pause`, `resume`, `isRunning`, `totalTime`, `timeSoFar`, and `reset`

Methods for manipulating timers in states: `timer`, `timerTime`, and `timerRunning`

`parseMessage` and `as.json` describe the JSON conversion system.

The functions `getOneRec` and `getManyRecs` use `parseEvent` to extract events from a database.

Examples

```
sw <- Timer("stopwatch")
swa <- parseTimer(as.jlist(sw,attributes(sw)))
stopifnot(isRunning(swa)==FALSE,
          timeSoFar(swa,as.POSIXct("2018-12-21 00:01"))==
          as.difftime(0,units="mins"),
          totalTime(swa)==as.difftime(0,units="mins"))

## Start the timer.
sw <- start(sw,as.POSIXct("2018-12-21 00:01"))
swa <- parseTimer(as.jlist(sw,attributes(sw)))
## Note that time so far is based on the time argument.
stopifnot(isRunning(swa)==TRUE,
          timeSoFar(swa,as.POSIXct("2018-12-21 00:10"))==
          as.difftime(9,units="mins"),
          totalTime(swa)==as.difftime(0,units="mins"))

## Pause the timer.
sw <- pause(sw,as.POSIXct("2018-12-21 00:10"))
swa <- parseTimer(as.jlist(sw,attributes(sw)))
stopifnot(isRunning(swa)==FALSE,
          timeSoFar(swa,as.POSIXct("2018-12-21 00:10"))==
          as.difftime(9,units="mins"),
          totalTime(sw)==as.difftime(9,units="mins"))
```

`Timer-class` *Class* "Timer"

Description

A `Timer` measures the time between events in an event sequence. Rather than containing an actual timer, it works by subtracting start and stop times from different events. Therefore “starting” a timer sets the start time to the current time (as measured by the most recent event), and reading the elapsed time at an event, looks at the difference between the start time and the stop time (as measured by the current event).

Objects from the Class

Objects can be created by calls of to the`Timer(...)` function. They are also created by the `Status` constructor when called with names for the timer objects.

Slots

`name`**:** Object of class `"character"` giving an identifier for the timer. Used in error reporting

`startTime`**:** Object of class `"POSIXct"`: the time at which the timer was started. If the timer is not running, this will be `NA`.

`totalTime`**:** Object of class `"difftime"`: the total elapsed time prior to the last start/resume call.

Methods

as.jlist `signature(obj = "Timer", ml = "list")`: converts the Timer object into a form to be serialized as a JSON object. See `as.jlist`.

isRunning `signature(timer = "Timer")`: returns `TRUE` if the timer is running and `FALSE` if not.

pause `signature(timer = "Timer", time = "POSIXct")`: Pauses the timer and sets the accumulated time to the elapsed time so far.

reset `signature(timer = "Timer")`: Pauses the timer and sets the accumulated time to zero.

resume `signature(timer = "Timer")`: Puts the timer back in the running state and does not affect the total elapsed time.

start `signature(timer = "Timer")`: Starts the timer.

timeSoFar `signature(timer = "Timer")`: Returns the elapsed time.

timeSoFar<- `signature(timer = "Timer")`: Sets the elapsed time.

totalTime `signature(timer = "Timer")`: Returns the current value of the total time field.

Details

The timer is not actually running a clock. It is instead counting the elapsed time between events. This primarily works by setting the `startTime` field when the timer is “started” and then differencing this from the current time as measured by the timestamp of the currently processed `Event`.

This should be transparent for most uses, with one note. Methods like `start`, `pause`, `resume`, and `timeSoFar` need to be passed the current time, so the `Timer` can adjust its internal state.

In the `getJS` and `setJS` functions, the timer behaves as if it has two virtual fields: `.run` or `.running` and `.time` or `.value`. The `.run` field returns the value of `isRunning`, and setting it to `FALSE` will cause the timer to `pause` and setting it to `TRUE` will cause the timer to `start` or `resume`. The `.time` or `.value` field returns the `timeSoFar` field of the timer. Setting it adjusts the `totalTime` without affecting the running state.

Note

For those who need a more detailed understanding, the timer works with two fields: `startTime` and `totalTime`. Intially `startTime` starts as `NA` and `totalTime` is 0. On a call to `start` or `resume`, the `startTime` is set to the current time. On a call to `pause` the difference between the `startTime` and the current time (passed as an argument) is added to the `totalTime`, and `startTime` is set to `NA`.

This means that `isRunning` is essentially `!is.na(startTime)`. If the timer is paused, the `timeSoFar` is the `totalTime`. If the timer is running, the `timeSoFar` is the `totalTime` plus the difference between the current time (passed as an argument) and the `startTime`. The setter "`timeSoFar<-`" will adjust `startTime` to the current `time` argument if the timer is running.

The `setJS` function gets the current time from the current `Event` object being processed, and thus automatically takes care of the time argument.

Author(s)

Russell Almond

See Also

The funciton `Timer` is the constructor, and the function `parseTimer` builds a `Timer` from JSON data.

The `Status` class contains a collection of timers. Other classes in the EIEvent system: `EIEngine`, `Context`, `Rule`, `Event`, `RuleTable`.

The functions `setJS`, `getJS` and `removeJS` have details about how to manipulate timers using rules.

Methods for working with Timers: `start`, `pause`, `resume`, `isRunning`, `totalTime`, `timeSoFar`, and `reset`

Methods for manipulating timers in states: `timer`, `timerTime`, and `timerRunning`

Examples

```
showClass("Timer")
```

`UserRecordSet-class` *Class* `"UserRecordSet"`

Description

A collection of user records associated with a given application; actually, a handle for the database collection holding the user records. User records are object of class `Status`.

Extends

All reference classes extend and inherit methods from `"envRefClass"`.

Fields

`app`: Object of class `character` giving the identifier for the application. This is part of the database key for the collection.

`dbname`: Object of class `character` giving the name of the database (e.g., “EIRecords”).

`dburi`: Object of class `character` giving the uri for the database, (e.g., “mongodb://localhost”).

`db`: Object of class `MongoDB` which is the handle to the database collection. As this field is initialized when first requested, it should not be accessed directly, but instead through the `recorddb()` method.

Class-Based Methods

`newStudent(uid)`: This generates a new blank status for a given student id. If there is an existing status for the student, that is returned. If not, if there is a status with uid “*DEFAULT*”, that is returned. If neither of those exists, a new blank status is created with context id “*INITIAL*”.

`getStatus(uid)`: Finds the current status for a given `uid`.

`initialize(app, dbname, dburi, db, ...)`: This sets up the object. Note that the db field is not initialized until `recorddb()` is first called.

`recorddb()`: This returns the handle to the `mongo` collection object. If the connection has not yet been initialized.

`saveStatus(state)`: This saves the status in the database.

Note

This is actually a wrapper for a database collection. The collection can contain records from many applications. The primary key for a record is `(app,uid,timestamp)`. For a given uid and application, the current record is the one with the most recent timestamp.

Author(s)

Russell G. Almond

References

The document “Rules Of Evidence” gives extensive documentation for the context system: `https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf`.

See Also

`Status`, `EIEngine`

Examples

```
showClass("UserRecordSet")
```

verb — *Accessor for verb and object field of events and rules.*

Description

Both `Event` and `Rule` objects have `verb`, `object` and `context` fields which describe the event that occurred or to which the rule applies. The vocabulary is given by the `app` field.

Usage

```
verb(x)
## S4 method for signature 'Event'
verb(x)
## S4 method for signature 'Rule'
verb(x)
object(x)
## S4 method for signature 'Event'
object(x)
## S4 method for signature 'Rule'
object(x)
context(x)
## S4 method for signature 'Event'
context(x)
## S4 method for signature 'Rule'
context(x)
## S4 method for signature 'Status'
context(x)
context(x) <- value
## S4 replacement method for signature 'Status'
context(x) <-value
oldContext(x)
## S4 method for signature 'Status'
oldContext(x)
```

Arguments

`x`	An object of class `Event` or `Rule` (or in the case of `context`, a `Status` object).
`value`	A character scalar giving a new value for the context.

Details

The `verb` and `object` fields are fairly self-explanatory. They are simply strings which give the verb and direct object of the activity in the presentation process which triggered the event. In the case of an `Event`, these are created by the presentation process. For `Rule` objects, the fields indicate the type of event for which the rule should be applicable. In particular, a rule is applicable to a certain event when the `verb` and `object` fields match. Note that if the the `verb` or `objec`field of thas the special keywork special keyword "ALL" that indicates that the rule is applicable for all verbs or objects respectively.

The context field is a little more complex. First, the value of the context field could correspond to a Context object or, in the case of a rule, be the keyword "ALL". For the Status (or Event) class this should be a primative context. In the case of a Rule class, it could also be a context group, which covers several other contexts through the belongsTo field. The rule context must match the context field in the Status (not the Event. The rule is considered applicable if (a) the contexts match exactly, (b) the rule context is a context set and the status context belongs to it, or (c) the rule context is the keyword "ALL".

In some cases, the presentation process will recognize and record the context. In this case the context field of the Event object will be of length 1. Otherwise, it will have the value character(0). In many cases, a change of context is recognized by a "Context" rule. After the context changes, the oldContext field retains the value of the previous context (useful for "Trigger" rules which usually fire after a change in context). The oldContext field gets reset when the EIEngine starts processing a new event.

Value

A character scalar giving the verb, object, or context.

Note

The xAPI format (Betts and Smith, 2018) uses "verb"s and "object"s, but they are much more complex objects. In EIEvent, the verb and object vocabularies are driven by the app field of the Event, and all supporting details are put in the details field.

Author(s)

Russell Almond

References

The document "Rules Of Evidence" gives extensive documentation for the JSON layout of the Event objects. https://pluto.coe.fsu.edu/Proc4/RulesOfEvidence.pdf.

Almond, R. G., Shute, V. J., Tingir, S. and Rahimi, S. (2018). Identifying Observable Outcomes in Game-Based Assessments. Talk given at the *2018 Maryland Assessment Research Conference*. Slides: https://education.umd.edu/file/11333/download?token=kmOIVIwi, Video: https://pluto.coe.fsu.edu/Proc4/Almond-Marc18.mp4.

Betts, B, and Smith, R. (2018). The Leraning Technology Manager's Guid to xAPI, Second Edition. HT2Labs Research Report: https://www.ht2labs.com/resources/the-learning-technology-managers-guide-to-
#gf_26.

See Also

Objects with verb and object fields: Event, Rule

Related fields of the event object. app details

The functions setJS, getJS and removeJS provide mechanisms for accessing the fields of an event object from Rule Conditions and Predicates.

Examples

```
ev2 <- Event("Phred","wash","window",
      timestamp=as.POSIXct("2018-12-21 00:02:01"),
      details=list(condition="streaky"))

stopifnot(verb(ev2)=="wash", object(ev2)=="window", context(ev2)==character(0))

r1 <- Rule(name="Coin Rule",
           doc="Set the value of the badge to the coin the player earned.",
           app="ecd://coe.fsu.edu/PPtest",
           verb="satisfied", object="game level",
           context="ALL",
           ruleType="Observable", priority=5,
           condition=list("event.data.badge"=c("silver","gold")),
           predicate=list("!set"=c("state.observables.badge"=
                                   "event.data.badge")))
stopifnot(verb(r1)=="satisfied", object(r1)=="game level",
          context(r1)=="ALL")

st <- Status("Phred","Level 0",timerNames=c("watch","stopwatch"),
   flags=list("list"=list("one"=1,"two"="too"),"vector"=(1:3)*10),
   observables=list("numeric"=12.5,char="foo",
                "list"=list("one"="a","two"=2),"vector"=(1:3)*100),
   timestamp=as.POSIXct("2018-12-21 00:01"))

stopifnot(context(st)=="Level 0",oldContext(st)=="Level 0")
context(st) <- "Level 1"
stopifnot(context(st)=="Level 1",oldContext(st)=="Level 0")
```

Index