The Mecate Package

Running a Bosal Generated Test Harness

Legal Notices and Information

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	July 16, 2018	Initial coding.	GAM
1.0	October 23, 2018	Initial release.	GAM
1.0.1	February 1, 2019	Corrected bug in delayed signal method.	GAM
1.1	February 2, 2019	Added procedures to format individual traces.	GAM

Contents

List of Figures

Introduction

This document is about a Tcl package called, `mecate`, which is used to interface with `micca` translated xUML domains running in a `bosal` generated test harnesses. This document is also a literate program and contains all the design information and code for the `mecate` package. Readers unfamiliar with literate programs should consult the appendix for details of the literate program syntax used in this document.

`Bosal` is an automatic test harness generator for `micca` domains. The test harness is an executable program that wraps one or more domains and provides a means to communicate with the outside world and allow the contained domains to be exercised. The test harness generated by `bosal` enables a `micca` domain to be compiled to run in a POSIX environment and to interact with the domain by driving its execution via operations and events. The harnessed domain also produces output containing information about state transitions, execution instrumentation information inserted by `micca`, and fatal errors detected at run time.

The basic test workflow is:

- Translate an xUML model using `micca`.
- Build a test harness executable using `bosal`. The test harness consists of one or more translated domains along with any required bridges, external entity stubs, or other code that is to be integrated together. It is usually useful to compile the domain code with compiler options that enable measuring code coverage.
- Connect to the test harness using `mecate`. Send commands to the test application and observe the various traces and other outputs to verify the application execution. This may be done using the `tcltest` package or as an ordinary Tcl script.
- Run sufficient test cases to obtain the code and state transition coverage desired. State machine dispatch traces and other information may be saved for further analysis.

Although the communications interface provided by `bosal` generated harnesses is ASCII record oriented, the `mecate` package is intended to support the running of test cases by providing a simpler interface to the harnessed application and to provide services to monitor the output of the application and take actions when specific traces, etc. are received as responses from the test application.

The `mecate` package is organized as a single TclOO class called `rein` and a few ordinary utility procedures.

```
<<mecate commands>>=
::oo::class create ::mecate::rein {
    <<rein configuration>>
}
```

The `rein` class command is also exported as an ensemble command from the `mecate` namespace. An object of the `rein` class has methods to support the workflow discussed above. The following sections describe the various methods of the `rein` class and are grouped according to topics. In many cases the methods provide only a thin veneer over the underlying `bosal` communications protocol. Other methods and data in the class support collecting state machine traces and instrumentation output from a test harness.

Class Management

The methods in this section handle constructing and configuring objects of the `rein` class.

Constructor

`mecate rein create` *objname ?options?*

`mecate rein new` *?options?*

The constructor for `rein` takes a set of option / value pairs. The options are:

`-timeout <msec timeout>`
: the default time out value for synchronization operations. If no timeout option is given, the default is 2000 milliseconds.

Implementation

```
<<rein configuration>>=
constructor {args} {
    ::logger::import -all -namespace log mecate

    set timeout 2000
    foreach {option value} $args {
        switch -exact -- $option {
            -timeout {
                set timeout $value
            }
            default {
                my Throw "unknown option, \"$option\""
            }
        }
    }

    log::debug "-timeout option = \"$timeout\""

    my variable tracesReceived
    set tracesReceived [::struct::queue]

    my variable instrReceived
    set instrReceived [::struct::queue]

    my variable fatalReceived
    set fatalReceived [::struct::queue]

    my variable tracePrefix
    set tracePrefix {}
    my variable instrPrefix
    set instrPrefix {}
    my variable fatalPrefix
    set fatalPrefix {}

    namespace import ::ral::*
    namespace import ::ralutil::*

    <<rein data structures>>

    return
}
```

The configuration information for the class is used frequently enough that we just make it variables available to all methods. Note that the variable names are the same as the option names and this is significant.

```
<<rein configuration>>=
variable timeout connection
```

Tests

```
<<constructor tests>>=
test constructor-1.0 {
    Create rein object
} -setup {
} -cleanup {
} -body {
    mecate rein create ::apptest
} -result {::apptest}
```

Destructor

reinobj `destroy`

The destructor for `rein` objects closes any communications connection made to the harnessed application.

Implementation

```
<<rein configuration>>=
destructor {
    my disconnect

    my variable tracesReceived
    $tracesReceived destroy

    my variable instrReceived
    $instrReceived destroy

    my variable fatalReceived
    $fatalReceived destroy

    relvar constraint delete R1
    relvar unset Trace TransitionTrace PolymorphicTrace CreationTrace\
            Instrumentation Fatal
}
```

Cget

reinobj `cget` *option*

option is one of the options described for the constructor. The method returns the value of the given option.

The `cget` method may be used to obtain the values of the options that were set during construction.

Implementation

```
<<rein configuration>>=
method cget {option} {
    switch -exact -- $option {
        -timeout {
            return [set [string range $option 1 end]]
        }

        default {
            my Throw "unknown option, \"$option\""
        }
    }
}
```

Tests

```
<<constructor tests>>=
test cget-1.0 {
    Get timeout value
} -setup {
} -cleanup {
} -body {
    ::apptest cget -timeout
} -result {2000}
```

Configure

reinobj `configure` *?opt1 value1 opt2 value2 . . . ?*

The `configure` method may be used to set the values of the configuration options. Option names and values are those described for the constructor. Returns the empty string.

Implementation

```
<<rein configuration>>=
method configure {args} {
    foreach {option value} $args {
        switch -exact -- $option {
            -timeout {
                set [string range $option 1 end] $value
            }

            default {
                my Throw "unknown option, \"$option\""
            }
        }
    }

    return
}
```

Tests

```
<<constructor tests>>=
test configure-1.0 {
    get timeout value
} -setup {
} -cleanup {
    ::apptest configure -timeout 2000
} -body {
    ::apptest configure -timeout 3000
    ::apptest cget -timeout
} -result {3000}
```

Test Harness Management

The methods in this section are used to set up and take down a test harness program. It is necessary to start the execution of a test harness before is can be used. Sometimes this is accomplished outside of a test script, for example if the test harness is running under control of a debugger.

When a `bosal` generated test harness starts, it acts as a server, opening a TCP port on the local host and awaiting a connection. To communicate with the test harness program, it is necessary to establish a TCP client connection to the test harness acting as a server. The `rein` class provides methods to manage these actions directly from a test script.

Start

reinobj `start` *program ?delay?*

program
The name of the test harness program.

delay
The the number of milliseconds to wait before control is returned to the caller. This time is used to insure that the harness program has had enough time to establish its communication interface before allowing the test to proceed.

The `start` method is used to begin execution of the test harness program. Starting is separated out as a method because certain test situations will have the harness program running under a debugger which will be responsible for starting program execution.

The return value is the process ID of the harness program, although such a number is of little use in the Tcl world. When the test harness communications channel is destroyed, the harness program itself exits.

Implementation

```
<<rein configuration>>=
method start {program {delay 500}} {
    log::info "starting harness program, \"$program\""
    set progargs [auto_execok $program]
    if {$progargs eq {}} {
        my Throw "unknown program, \"$program\""
    }
    set pid [exec -ignorestderr -- $progargs &]
    after $delay
    return $pid
}
```

Tests

```
<<exported tests>>=
test start-1.0 {
    start test harness program
} -setup {
} -cleanup {
} -body {
    ::apptest start ./bstore_harness
} -result {[0-9]+} -match regexp
```

Tests

```
<<exported tests>>=
test start-2.0 {
    unknown program
} -setup {
} -cleanup {
} -body {
    ::apptest start ./foo
} -result {unknown program, "./foo"} -returnCodes error
```

Connect

reinobj `connect` *?port?*

The `connect` method forms a communications connection to the test harness. All I/O to and from the test harness happens across this connection.

`port`
The local host port number on which the test harness program accepts connection requests. By default this is port 3906.

Returns the empty string. Throws an error if the connection cannot be established.

Implementation

```
<<rein configuration>>=
method connect {{port 3906}} {
    log::info "connecting to localhost:$port"

    set tries 0                                                          ; # ❶
    while {true} {
        try {
            set connection [socket localhost $port]
            chan configure $connection -blocking true -buffering line
            chan event $connection readable [mymethod HandleHarnessInput]   ; # ❷
            return
        } on error {result opts} {
            incr tries
            if {$tries >= 3} {
                log::error "connection failure after $tries attempts"
                return -options $opts $result
            }
            log::notice "connection attempt failed, waiting 500 ms to try again"
            after 500
        }
    }
}
```

❶ We intend to try the connection 3 times, pausing between any failures.

❷ All the input is received by file `readable` events. This is described below.

Tests

```
<<exported tests>>=
test connect-1.0 {
    connect to program
} -setup {
} -cleanup {
} -body {
    ::apptest connect
} -result {}
```

Disconnect

reinobj `disconnect`

The `disconnect` method closes the communications connection to the test harness. This causes the test harness process to exit.

Implementation

```
<<rein configuration>>=
method disconnect {} {
    log::info "disconnecting from test harness"
    if {[info exists connection]} {
        catch {chan close $connection}
        unset -nocomplain connection
    }
    return
}
```

Null

reinobj `null`

The `null` method performs no action on the test domain, but communicates with it to insure that it is responsive.

Implementation

```
<<rein configuration>>=
method null {} {
    return [my Command null]                ; # ❶
}
```

❶ The `Command` method does all the heavy lifting when interacting with the test harness. This method is discussed below.

Tests

```
<<exported tests>>=
test null-1.0 {
    null method test
} -setup {
} -cleanup {
} -body {
    ::apptest null
} -result {}
```

Version

reinobj `version`

The `version` method returns the version number of `bosal` used to create the test harness code.

Implementation

```
<<rein configuration>>=
method version {} {
    return [my Command version]
}
```

Tests

```
<<exported tests>>=
test version-1.0 {
    version method test
} -setup {
```

```
} -cleanup {
} -body {
    ::apptest version
} -result {1.2}
```

Harness Introspection

A `bosal` generated harness has introspection capabilities that can be used to determine the characteristics of the domains in the harness. This information is available using the `query` method described in this section.

Query

reinobj `query` *what ?arg1 arg2 . . . ?*

`what`
The type of query to be performed. The `what` argument may be one of the following:

domains
Returns the names of the domains contained in the test harness.

operations ***domain***
Returns the names of the domain operations for the given domain.

doparams ***domain operation***
Returns the names and data types of the parameters for the given domain operation.

classes ***domain***
Returns the names of the class for the given domain.

attributes ***domain class***
Returns the names of the attributes for the given class.

instances ***domain class***
Returns the total number of instances and the name and identifiers for the pre-existing instances.

states ***domain class***
Returns the names of the states for the given class.

events ***domain class***
Returns the names of the events for the given class.

evparams ***domain class event***
Returns the parameter names and data types for the given event.

current ***domain class instance***
Returns the current state of the given class instance.

The `query` method returns the result of querying various entities in a test harness. It is possible to determine the domains in the harness, the classes of each domain and many other characteristics.

Implementation

```
<<rein configuration>>=
method query {what args} {
    return [my Command query $what {*}$args]
}
```

Tests

```
<<exported tests>>=
test query-1.0 {
    Query the domains in the harness
} -setup {
} -cleanup {
} -body {
    ::apptest query domains
} -result {bookstore}
```

```
<<exported tests>>=
test query-1.1 {
    Query the domain operations in the bookstore domain
} -setup {
} -cleanup {
} -body {
    set ops [::apptest query operations bookstore]
} -result {changeSelectionQuantity}
```

```
<<exported tests>>=
test query-1.2 {
    Query the domain operation parameters for changeSelectionQuantity
} -setup {
} -cleanup {
} -body {
    set ops [::apptest query doparams bookstore changeSelectionQuantity]
} -result {cartID MRT_InstId productID MRT_InstId quantity unsigned}
```

```
<<exported tests>>=
test query-2.0 {
    Query the classes in the bookstore domain
} -setup {
} -cleanup {
} -body {
    set classes [lsort [::apptest query classes bookstore]] ; # ❶
    return [lindex $classes 0]
} -result {Artist}
```

❶ Just pick off the first name, alphabetically, to test against.

```
<<exported tests>>=
test query-2.1 {
    Query the attributes of the Artist class
} -setup {
} -cleanup {
} -body {
    ::apptest query attributes bookstore Artist
} -result {artistName}
```

```
<<exported tests>>=
test query-2.2 {
    Query the instances of the Artist class
} -setup {
} -cleanup {
} -body {
    ::apptest query instances bookstore Artist
} -result {total 2 named {fs 1 mj 0}}
```

```
<<exported tests>>=
test query-3.0 {
    Query the states of the ShoppingCart class
} -setup {
} -cleanup {
} -body {
    ::apptest query states bookstore ShoppingCart
} -result {@ NewOrder AddingSelectiontoOrder CancelingEntireOrder\
        EstablishingCustomerandVerifyingPayment}
```

```
<<exported tests>>=
test query-4.0 {
    Query the events of the ShoppingCart class
} -setup {
} -cleanup {
} -body {
    ::apptest query events bookstore ShoppingCart
} -result {addSelection cancel checkOut startCart}
```

```
<<exported tests>>=
test query-4.1 {
    Query the event parameters of the startCart event
} -setup {
} -cleanup {
} -body {
    ::apptest query evparams bookstore ShoppingCart startCart
} -result {productID {char const *} quantity unsigned}
```

```
<<exported tests>>=
test query-5.0 {
    Query the current state of an instance
} -setup {
} -cleanup {
} -body {
    ::apptest query current bookstore Warehouse acme
} -result {WaitingforaShipment}
```

Operations on Domains

Domainop

reinobj `domainop` *operation ?arg1 arg2 ... ?*

`operation`
The name of the domain operation to invoke.

`argN`
Arguments to the domain operation. The order of the arguments must be the same as that defined for the given domain operation.

The `domainop` method invokes a domain operation on a domain.

Implementation

```
<<rein configuration>>=
method domainop {domain opname args} {
    return [my Command domainop $domain $opname {*}$args]
}
```

Operations on Class Instances

Create

reinobj `create` *domain class*

`domain`
The name of the domain containing the class.

`class`
The name of the class of the instance to be create.

The `create` method creates an instance of the given class and returns a integer identifier for the instance. The identifying number is unique only within the class.

N.B. that the instance is created in an uninitialized state and it is necessary to follow up with invocation of the `update` method on the instance to initialize it attributes. Also, creating instances can cause problems with referential integrity which will be detected on the next transaction boundary. Since there is no way to deal with associations from the test harness interface, creating class instances is of limited utility.

Implementation

```
<<rein configuration>>=
method create {domain class} {
    return [my Command create $domain $class]
}
```

Delete

reinobj `delete` *domain class inst*

`domain`
The name of the domain containing the instance.

`class`
The name of the class to which the instance belongs.

`inst`
The integer identifier of the instance. Pre-existing instances may use the instance name for this field.

The `delete` method deletes the given instance.

N.B. Deleting an instance often causes a referential integrity problem that my create a fatal error when the next thread of control is finished. In general, it is not safe to delete arbitrary instances and there is no mechanism provide to manipulate relationship instances in the harnessed domains.

Implementation

```
<<rein configuration>>=
method delete {domain class inst} {
    return [my Command delete $domain $class $inst]
}
```

Read

reinobj `read` *domain class inst _?attr1 attr2 . . . ?*

`domain`
The name of the domain in which the instance is defined.

`class`
The name of the class to which the instance belongs.

`inst`
The name or ID number of the class instance to read.

`attrN`
The names of the attributes whose values are returned. If no attribute names are given, then the return value is a list of alternating attribute name / attribute value elements for all the attributes of the class. Otherwise, only the named attribute values are returned as name / value elements.

The `read` method reads the values of attributes from classes in a domain.

N.B. Not all attributes of an instance may be read. In particular, instance data used to implement associations and generalizations is not available since these values are memory addresses and not meaningful outside of the domain.

Implementation

```
<<rein configuration>>=
method read {domain class inst args} {
    return [my Command read $domain $class $inst {*}$args]
}
```

Tests

```
<<exported tests>>=
test read-1.0 {
    Read the value of the product ID for the XUML book.
} -setup {
} -cleanup {
} -body {
    ::apptest read bookstore Product mbxuml productID
} -result {productID 14387}
```

Update

reinobj `update` *domain class inst _?attr1 value1 attr2 value2 . . . ?*

`domain`
The name of the domain in which the instance is defined.

`class`
The name of the class to which the instance belongs.

`inst`
The name or ID number of the class instance to read.

`attrN`
The name of the attribute to whose value is to be updated..

`valueN`
The corresponding value that is to be updated into the attribute.

The `update` method updates the values of attributes of classes in a domain. The attribute names and values are given in pairs as arguments to the method. Note that multiple attributes may be updated in a single invocation of the method.

Implementation

```
<<rein configuration>>=
method update {domain class inst args} {
    return [my Command update $domain $class $inst {*}$args]
}
```

Tests

```
<<exported tests>>=
test update-1.0 {
    Update the value of the email of the mjb author
} -setup {
} -cleanup {
} -body {
    ::apptest update bookstore Author mjb email mjb@gmail.com
} -result {email mjb@gmail.com}
```

Signaling Events

One of the primary ways to cause a domain to react is to signal events to the class instances. Both immediate and delayed signaling is supported.

Signal

Signals generated from a test harness are considered to be generated outside of a state machine context and so start a new thread of control

reinobj `signal` *domain class inst event ?param1 param2 . . . ?*

`domain`
The name of the domain to which the signaled instance belongs.

`class`
The name of the class to which the signaled instance belongs.

`inst`
The integer identifier or name of the instance to signal.

`event`
The name of the event to signal.

`paramN`
Parameters of the event. Parmeters must be given in the same order as defined for the event or state into which the event causes a transition.

The `signal` method signals an event to a class instance.

Implementation

```
<<rein configuration>>=
method signal {domain class inst event args} {
    return [my Command signal $domain $class $inst $event {*}$args]
}
```

Delayed Signal

Delayed signals are a request for the system to issue the signal at some minimum time in the future.

reinobj `delaysignal` *domain class inst delay event ?param1 param2 . . . ?*

`domain`
The name of the domain to which the signaled instance belongs.

`class`
The name of the class to which the signaled instance belongs.

`inst`
The integer identifier or name of the instance to signal.

`delay`
The minimum number of milliseconds that are to elapse before the event is signaled. A delay value of 0 results in the event being signaled immediately.

`event`
The name of the event to signal.

`paramN`
Parameters of the event. Parmeters must be given in the same order as defined for the event or state into which the event causes a transition.

The `delaysignal` method signals an event to a class instance at some time in the future.

Implementation

```
<<rein configuration>>=
```

```
method delaysignal {domain class inst delay event args} {
    return [my Command delaysignal $domain $class $inst $delay $event {*}$args]
}
```

Cancel Signal

reinobj `cancel` *domain class inst event*

`domain`
The name of the domain of the instance whose event is to be canceled.

`class`
The name of the class of the instance whose event is to be canceled.

`inst`
The instance identifier or instance name whose event is to be canceled.

`event`
The name of the event to cancel.

The `cancel` method requests an delayed signal be canceled.

Implementation

```
<<rein configuration>>=
method cancel {domain class inst event} {
    return [my Command cancel $domain $class $inst $event]
}
```

Remaining Delay for a Signal

reinobj `remaining` *domain class inst event*

`domain`
The name of the domain for the instance whose remaining time is requested.

`class`
The name of the class of the instance whose remaining time is requested.

`inst`
The instance identifier or instance name remaining time is requested.

`event`
The name of the event for which the remaining delay time is requested.

The `remaining` method returns the number of milliseconds before the event is to be signaled. A return value of 0, implies that either the event has already been signaled (*i.e.* its delay time has already expired) or the delayed event did not exist.

Implementation

```
<<rein configuration>>=
method remaining {domain class inst event} {
    return [my Command remaining $domain $class $inst $event]
}
```

Create async

Asynchronous creation of instances happens by sending a creation event. A new instance is created and an event is dispatched to cause a transition in the newly created instance.

reinobj `createasync` *domain class event ?param1 param2 . . . ?*

`domain`
The name of the domain in which the class resides.

`class`
The name of the class for which an instance is to be created.

`event`
The name of the creation event to be signaled.

`paramN`
The event parameters for the creation event.

The `createasync` method creates a class instance asynchronously by signaling a creation event.

Implementation

```
<<rein configuration>>=
method createasync {domain class event args} {
    return [my Command createasync $domain $class $event {*}$args]
}
```

Controlling Execution

The test harness generated by `bosal` makes extensive use of the means provided by the `micca` run-time code to control the event loop. The `micca` run-time code allows the dispatch of individual events, an entire thread of control and exiting the event event loop. In addition, the test harness installs its own fatal condition handler to prevent the harness program from exiting if a fatal error condition arises.

Event Loop

reinobj `eloop` *?run | halt | once | toc ?wait | nowait? ?*

With no arguments the `eloop` method returns the current state of the event loop, either **running** or **halted**.

An additional argument may be given to control the state of the event loop.

run
Start running the event loop. Returns **running**.

halt
Halt the dispatching of events. Returns **halted**.

once
Dispatch one event from the event loop. Returns **true** if the event was dispatched and **false** if no event was awaiting dispatch.

toc ?wait | nowait?
Dispatch one thread of control. If **wait** is specified (or if no additional option is given), then the event loop will wait until a thread of control event arrives. If **nowait** is given and if there is no pending thread of control event, the command returns immediately. Returns **true** if a thread of control was dispatched and **false** otherwise.

Implementation

```
<<rein configuration>>=
method eloop {args} {
    return [my Command eloop {*}$args]
}
```

Tests

```
<<exported tests>>=
test eloop-1.0 {
    Query execution state
} -setup {
} -cleanup {
} -body {
    ::apptest eloop
} -result {running}
```

```
<<exported tests>>=
test eloop-2.0 {
    Halt execution
} -setup {
} -cleanup {
} -body {
    ::apptest eloop halt
} -result {halted}
```

```
<<exported tests>>=
test eloop-3.0 {
    Dispatch an event
} -setup {
} -cleanup {
    ::apptest eloop run ; # set things back to running for later
} -body {
    ::apptest eloop once
} -result {false}
```

State Machine Traces

For programs using state machines that are implemented in a callback manner, a chronological trace of the dispatched events is essential for effective debugging. The `micca` run-time code can produce this trace and the `bosal` generated harness formats and transmits the trace information.

In `mecate`, we have four points of control over state machine traces.

1. Tracing can be turned on or off. If not needed, then the traces can be turned off.
2. Synchronizing script execution with a state machine trace. Facilities are provided to suspend a script's execution until either a given trace is received or a timeout occurs.
3. Trace accumulation and formatting. All traces can be accumulated, saved and formatted into human readable strings. Traces can also be rendered as a UML sequence diagram using an external program named, `plantuml`.
4. Trace notification. Traces can be forwarded via a callback when they arrive.

N.B. that tracing can be compiled out of the `micca` generated code. If an object file obtained by compiling the domain code file with the pre-processor symbol, `MRT_NO_TRACE`, defined is linked into a test harness, then no trace information will be available for that domain.

Trace

reinobj `trace` *?on | off?*

`on | off`
Turn tracing either **on** or **off**. In all cases, the current state of the tracing is returned.

The `trace` method controls whether state machine dispatch traces are output from the test harness and accumulated by `mecate`.

Implementation

```
<<rein configuration>>=
method trace {args} {
    return [my Command trace {*}$args]
}
```

Tests

```
<<exported tests>>=
test trace-1.0 {
    Query tracing
} -setup {
} -cleanup {
} -body {
    ::apptest trace
} -result {off}
```

```
<<exported tests>>=
test trace-1.1 {
    Turn tracing on
} -setup {
} -cleanup {
} -body {
    ::apptest trace on
} -result {on}
```

```
<<exported tests>>=
test trace-1.2 {
    Turn tracing off
} -setup {
} -cleanup {
} -body {
    ::apptest trace off
} -result {off}
```

Wait for Event Trace

A common test sequence is to signal an event and then examine the attributes of a class after the event has been dispatched. This implies that it is necessary to know when the event has been dispatched. We want to suspend test execution until the effect of signaling the event has happened so we can stay synchronized to the test harness execution.

To accomplish this, each state machine event trace that is received by `mecate` is placed in a queue that is used as a cache for the traces. The test code may then examine the traces in the queue and if a match is found, know that event has been dispatched. If no match is found in the cache of traces, then it is necessary to wait for other traces to arrive, assuming the matching event dispatch happens in the future.

reinobj `waitForEventTrace` *?field1 pattern1 field2 pattern2?*

`fieldN`
The name of a field of the event trace. Valid field names are:

`type`
The type of the transition. The value of the type field is either **transition**, **polymorphic**, or **creation**.

`event`
The name of the event.

`source`
The class instance that signaled the event.

`target`
The class instance that received the event.

`time`
A timestamp of when the event was dispatched.

For **transition** type events, the following fields are also available:

`currstate`
The name of the state before the event dispatch.

`newstate`
The new of the state after the event dispatch. If this field is `IG`, the event was ignored. If this field is `CH`, the event caused an error condition.

For **polymorphic** type events, the following fields are also available:

`relationship`
The name of the generalization relationships across which the polymorphic event is inherited.

`newevent`
The new name the event has in the context of the next level of the generalization.

`subclass`
The name of the subclass to which the superclass instance was related when the polymorphic event was dispatched.

For **creation** type events, there are no additional fields.

`patternN`
The corresponding pattern to match for the given key. Patterns may be of any form accepted by the `string match` command.

The `waitForEventTrace` method examines the cache of state machine event traces in chronological order searching for a match to the field / pattern pairs given. Any trace that does not match is discarded from the cache. If necessary, execution of the test is suspended until a state machine trace matching the given parameters arrives or a timeout occurs. Matching multiple fields is conjunctive in nature, *i.e.* if multiple fields are given, then *all* fields must match to consider the event trace to be a match.

The method returns the matching state machine trace. If a timeout or fatal error occurs, then the method throw an error.

Implementation

```
<<rein configuration>>=
method waitForEventTrace {args} {
    my variable tracesReceived
    my variable traceSyncVar

    set expect $args
    while {true} {
```

```
        while {[$tracesReceived size] != 0} {
            set actual [$tracesReceived get]

            if {[my MatchTrace $actual $expect]} {
                return $actual
            } else {
                log::debug "discarding trace, \"$actual\":\
                        failed to match, \"$expect\""
            }
        }

        set resolvedVarName [my varname traceSyncVar]
        set timer [expr {$timeout > 0 ?\
            [after $timeout [list set $resolvedVarName TIMEOUT]] : {}}]
        vwait $resolvedVarName

        if {$traceSyncVar eq "TIMEOUT"} {
            my Throw "timed out on receiving event traces"
        } elseif {$traceSyncVar eq "FATAL"} {
            my Throw "fatal run-time error"
        } else {
            if {$timer ne {}} {
                after cancel $timer
            }
        }
    }
}
```

The `MatchTrace` method compares a trace against a pattern and determines if there is a match. The matches performed on traces can be applied to all types of traces and so is factored out here. Since the traces are actually dictionaries, the code is quite generic.

Implementation

```
<<rein configuration>>=
method MatchTrace {trace pattern} {
    set matchCount 0
    dict for {key value} $pattern {
        if {[dict exists $trace $key] &&\
                [string match $value [dict get $trace $key]]} {
            incr matchCount
        }
    }

    return [expr {$matchCount == [dict size $pattern] ? "true" : "false"}]
}
```

Clear the Event Trace Cache

reinobj `clearEventTraceCache`

The `clearEventTraceCache` method deletes any accumulated state machine event traces contained in the event trace cache. All future attempts to match an event trace will operate only against those traces received after this method was invoked.

Implementation

```
<<rein configuration>>=
method clearEventTraceCache {} {
    my variable tracesReceived
```

```
    while {[$tracesReceived size] != 0} {
        $tracesReceived get
    }
    return
}
```

Format Event Traces

reinobj `formatEventTraces` *?begin? ?finish?*

`begin`
An optional number giving the starting trace to include in the output. If missing, then `begin` is taken as 1, the first trace accumulated.

`finish`
An optional number giving the last trace to include in the output. If missing, then `finish` is taken as the last trace accumulated.

The `formatEventTraces` method returns a string containing a human readable print out of the received traces.

Implementation

```
<<rein configuration>>=
method formatEventTraces {{begin 1} {finish end}} {
    set traces [pipe {
        my RestrictTraces $begin $finish Trace |
        rvajoin ~ [relvar set TransitionTrace] Transition |
        rvajoin ~ [relvar set PolymorphicTrace] Polymorphic |
        rvajoin ~ [relvar set CreationTrace] Creation
    }]                                                      ; # ❶

    set listing [list]                                      ; # ❷
    relation foreach trace $traces -ascending TraceId {
        relation assign $trace
        set line {}
        append line [format "% 3u " $TraceId]
        if {[relation isnotempty $Transition]} {
            relation assign $Transition Current New

            append line [format\
                "%s: Transition: %s - %s -> %s: %s ==> %s"\
                $Time $Source $Event $Target $Current $New\
            ]
        } elseif {[relation isnotempty $Polymorphic]} {
            relation assign $Polymorphic Relationship NewEvent Subclass

            append line [format\
                "%s: Polymorphic: %s - %s -> %s: %s - %s -> %s"\
                $Time $Source $Event $Target $Relationship $NewEvent $Subclass\
            ]
        } elseif {[relation isnotempty $Creation]} {
            append line [format\
                "%s: Creation: %s - %s -> %s"\
                $Time $Source $Event $Target\
            ]
        }

        lappend listing $line
```

```
    }

    return [join $listing "\n"]
}
```

❶ This query joins the selected traces with the three subclasses of traces based on the trace type. Because of the properties of a *partition constraint* exactly one of the relation valued attributes from the `rvajoin` operations (*i.e.* either **Transition**, **Polymorphic**, or **Creation**) will be non-empty and the non-empty attribute will contain exactly one tuple. This is just a convenient way to deal with the navigation of the partition constraint when iterating across all the tuples in the relvar.

❷ The strategy is to accumulate each formatted trace into a list and then join the list elements to create a printable string.

We will find other uses for a general method to select a subset of trace information to handle. So it is factored out here.

```
<<rein configuration>>=
method RestrictTraces {begin finish which} {
    if {$begin == 1 && $finish eq "end"} {                  ; # ❶
        set traces [relvar set $which]
    } else {
        if {$finish eq "end"} {
            set finish [relation cardinality [relvar set $which]]
        } elseif {![string is integer $finish]} {
            my Throw "bad finish value, \"$finish\""
        }
        set traces [relation restrictwith [relvar set $which]\
                {$TraceId >= $begin && $TraceId <= $finish}]
    }
    return $traces
}
```

❶ First, we need to select the traces that have been requested. The default arguments select all of the traces. Otherwise, we need to find the requested subset. To save iterating over all the traces, we check if the default arguments have been passed.

Discard Event Traces

reinobj `discardEventTraces`

The `discardEventTraces` method deletes the set of accumulated event traces.

Implementation

```
<<rein configuration>>=
method discardEventTraces {} {
    variable traceId
    set traceId 0                   ; # ❶
    relvar eval {
        relvar set Trace [relation empty [relvar set Trace]]
        relvar set TransitionTrace [relation empty [relvar set TransitionTrace]]
        relvar set PolymorphicTrace [relation empty [relvar set PolymorphicTrace]]
        relvar set CreationTrace [relation empty [relvar set CreationTrace]]
    }                               ; # ❷
    return
}
```

❶ Event trace numbering is also reset.

❷ Clearing the trace relvars must be done in a transaction because of the **R1** partition constraint.

Trace Notification

reinobj `traceNotify` *?prefix?*

`prefix`
A command prefix.

The `traceNotify` method returns and optionally sets the command prefix for a command that is executed whenever a state machine trace arrives from a test harness. The `prefix` argument is optional. If present, then the argument is interpreted as a command prefix and will be invoked with an additional argument as: `prefix` *trace* where *trace* is a dictionary with the trace information. The keys to the trace dictionary are the same as the field names listed in the `waitForEventTrace` method. If missing, then no changes are made to the notification command. In both cases, the current value of the notification command prefix is returned. Setting the *prefix* to the empty string, disables trace notification.

Implementation

```
<<rein configuration>>=
method traceNotify {args} {
    my variable tracePrefix
    if {[llength $args] == 1} {
        set tracePrefix [lindex $args 0]
    } elseif {[llength $args] > 1} {
        my Throw "wrong number of arguments"
    }
    return $tracePrefix
}
```

We must also provide a command that the event handler can use to notify when a trace arrived.

```
<<rein configuration>>=
method NotifyTraceArrived {trace} {
    my variable tracePrefix
    if {$tracePrefix eq {}} {
        return
    }

    try {
        eval [linsert $tracePrefix end $trace]
    } on error {result} {
        log::error "error notifying trace arrival: \"$result\""
        set tracePrefix {}                                          ; # ❶
    }
}
```

❶ Notice we turn off trace arrival notification if there was an error. This prevents repeated error messages.

Event Trace Format

`mecate eventTraceFormat` *trace*

`trace`
A state machine event trace in dictionary form as returned from the test harness.

The `eventTraceFormat` procedure returns a string containing a human readable version of the *trace*.

Implementation

```
<<mecate commands>>=
proc ::mecate::eventTraceFormat {trace} {
    dict with trace {
        switch -exact -- $type {
            transition {
                return [format\
                    "%s: Transition: %s - %s -> %s: %s ==> %s"\
                    $time $source $event $target $currstate $newstate\
                ]
            }
            polymorphic {
                return [format\
                    "%s: Polymorphic: %s - %s -> %s: %s - %s -> %s"\
                    $time $source $event $target $relationship $newevent\
                    $subclass\
                ]
            }
            creation {
                return [format\
                    "%s: Creation: %s - %s -> %s"\
                    $time $source $event $target\
                ]
            }
            default {
                my Throw "unknown trace type, \"$type\""
            }
        }
    }
}
```

Sequence Diagram of State Machine Traces

Although a textual list of state machine traces is useful, visualizing the interactions of state machines is often easier if the trace information is displayed in graphical form. The UML sequence diagram can be used for this purpose.

`Mecate` does not perform the graphical rendering of the sequence diagram. That task is delegated to a program named, `plantuml`. `Plantuml` accepts a text file describing the instances and events in the sequence diagram and it perfrom the layout and graphical generation. The role of `mecate` is to format the state machine traces into text that is acceptable to `plantuml`.

reinobj `seqDiag` *?begin? ?finish?*

`begin`
An optional number giving the starting trace to include in the output. If missing, then `begin` is taken as 1, the first trace accumulated.

`finish`
An optional number giving the last trace to include in the output. If missing, then `finish` is taken as the last trace accumulated.

The `seqDiag` method returns a string consisting of text that can be given to `plantuml` to create a graphical representation of the state machine traces as a UML sequence diagram.

When `planuml` lays out a sequence diagram, the order of defining the participants is important. `Planuml` places the race track headings for the instances in a left to right order as they are declared. To keep the general flow of the diagram to be left to right, we want to declare the race trace headings to `plantuml` in the same sequence as they occur in the set of traces being displayed. The complication comes in that an instance heading may first occur as either a **Source** of an event or as the **Target** of an event.

Implementation

```
<<rein configuration>>=
method seqDiag {{begin 1} {finish end}} {
    lappend uml "@startuml"

    set traces [my RestrictTraces $begin $finish Trace]

    <<seqDiag: declare participants in event order>>

    set traces [pipe {
        rvajoin $traces [relvar set TransitionTrace] Transition |
        rvajoin ~ [relvar set PolymorphicTrace] Polymorphic |
        rvajoin ~ [relvar set CreationTrace] Creation
    }]

    relation foreach trace $traces -ascending TraceId {
        relation assign $trace

        if {$Source eq "?.?"} {
            set Source $extlabel
        }

        if {[relation isnotempty $Transition]} {
            lappend uml "\"$Source\" --> \"$Target\" : $Event"
        } elseif {[relation isnotempty $Creation]} {
            lappend uml "create \"$Target\""
            lappend uml "\"$Source\" -> \"$Target\" : $Event <<create>>"
        } elseif {[relation isnotempty $Polymorphic]} {
            relation assign Polymorphic
            lappend uml "\"$Source\" --> \"$Target\" :\
                    $Event <<poly \[$Relationship\]>>"
        }
    }

    lappend uml "@enduml"

    return [join $uml "\n"]
}
```

The strategy to find all the participants is to project both the **Source** and **Target** of an event and take the union of the two projections. To insure that a **Target** appears after a **Source**, we will add one to its **TraceId**. The **TraceId** attribute is then used to order the result.

```
<<seqDiag: declare participants in event order>>=
set srcInsts [pipe {
    relation project $traces TraceId Source |
    relation rename ~ Source Name
}] ;                                                                    # ❶
set trgInsts [pipe {
    relation project $traces TraceId Target |
    relation rename ~ Target Name |
    relation update ~ t {1} {
        tuple update $t TraceId [expr {[tuple extract $t TraceId] + 1}]}
}] ;                                                                    # ❷
set insts [pipe {
    relation union $srcInsts $trgInsts |
    relation summarizeby ~ Name s First int {rmin($s, "TraceId")} |
    relation list ~ Name -ascending First
}] ;                                                                    # ❸

set extlabel BOUNDARY
foreach inst $insts {
```

```
    if {$inst eq "?.?"} {
        lappend uml "boundary \"$extlabel\"" ;                          # ❹
    } else {
        lappend uml "participant \"$inst\""
    }
}
```

❶ We change the name of the **Source** attribute to be **Name** to enable the union operation below.

❷ The `update` operation adds 1 to the **TraceId** attribute for **Target** instances.

❸ The `summarizeby` operation finds the minimum **TraceId** value in a group of names. That **TraceId** is the first event where the instances is mentioned. Creating a list of names in ascending order of first appearance gives us the order we want `plantuml` to display the race track headings.

❹ We use a label more informative than "?.?" to show events coming from outside of a state machine context.

reinobj `seqDiagToChan` *channel ?begin? ?finish?*

`channel`
The name of a Tcl I/O channel that is writable.

`begin`
An optional number giving the starting trace to include in the output. If missing, then `begin` is taken as 1, the first trace accumulated.

`finish`
An optional number giving the last trace to include in the output. If missing, then `finish` is taken as the last trace accumulated.

The `seqDiagToChan` method writes to a Tcl channel a string consisting of text that can be given to `plantuml` to create a graphical representation of the state machine traces as a UML sequence diagram.

Implementation

```
<<rein configuration>>=
method seqDiagToChan {channel {begin 1} {finish end}} {
    chan puts $channel [my seqDiag $begin $finish]
    return
}
```

reinobj `seqDiagToFile` *filename ?begin? ?finish?*

`filename`
The name of a file to which the sequence diagram text is to be written.

`begin`
An optional number giving the starting trace to include in the output. If missing, then `begin` is taken as 1, the first trace accumulated.

`finish`
An optional number giving the last trace to include in the output. If missing, then `finish` is taken as the last trace accumulated.

The `seqDiagToFile` method writes to a file a string consisting of text that can be given to `plantuml` to create a graphical representation of the state machine traces as a UML sequence diagram.

Implementation

```
<<rein configuration>>=
method seqDiagToFile {file {begin 1} {finish end}} {
    set channel [open $file w]
    try {
        my seqDiagToChan $channel $begin $finish
    } finally {
        chan close $channel
    }
    return
}
```

Instrumentation Traces

In addition to event traces, `micca` domains produce instrumentation traces. During code generation, `micca` inserts trace statements into state activities and other functions. The traces statements are actually "C" pre-processor macros which, by default, produce instrumentation output of the function name, the file containing the function, and line number. This gives a record which functions in the the domain are executed. In addition, the `MRT_DEBUG` macro, which has the same interface as `printf()`, is available to activities for generating instrumentation output.

In a test harness, the instrumentation output is directed to the harness communications port and the methods in this section define the operations that can be applied to instrumentation traces.

Instr

reinobj `instr` *?on | off?*

`on | off`
: Turn tracing either **on** or **off**. In all cases, the command returns the current state of instrumentation tracing.

The `instr` method controls whether instrumentation traces are output from the test harness and accumulated.

Implementation

```
<<rein configuration>>=
method instr {args} {
    return [my Command instr {*}$args]
}
```

Tests

```
<<exported tests>>=
test instr-1.0 {
    Query instrumentation
} -setup {
} -cleanup {
} -body {
    ::apptest instr
} -result {off}
```

Wait for Instrumentation Trace

reinobj `waitForInstrTrace` *?field1 pattern1 field2 pattern2?*

`fieldN`
The name of a field of the instrumentation trace. Valid field names are:

`time`
A timestamp of when the trace was generated.

`message`
The text of the instrumentation trace message.

`patternN`
The corresponding pattern to match for the given key. Patterns may be of any form accepted by the `string match` command.

The `waitForInstrTrace` method suspends execution until a instrumentation trace matches the given parameters or a timeout occurs.

Implementation

```
<<rein configuration>>=
method waitForInstrTrace {args} {
    my variable instrReceived
    my variable instrSyncVar

    set expect $args
    while {true} {
        while {[$instrReceived size] != 0} {
            set actual [$instrReceived get]

            if {[my MatchTrace $actual $expect]} {
                return $actual
            } else {
                log::debug "discarding trace, \"$actual\":\
                        failed to match, \"$expect\""
            }
        }

        set resolvedVarName [my varname instrSyncVar]
        set timer [expr {$timeout > 0 ?\
            [after $timeout [list set $resolvedVarName TIMEOUT]] : {}}]
        vwait $resolvedVarName

        if {$instrSyncVar eq "TIMEOUT"} {
            my Throw "timed out on receiving event traces"
        } elseif {$instrSyncVar eq "FATAL"} {
            my Throw "fatal run-time error"
        } else {
            if {$timer ne {}} {
                after cancel $timer
            }
        }
    }
}
```

Clear the Instrumentation Trace Cache

reinobj `clearInstrTraceCache`

The `clearInstrTraceCache` method deletes any accumulated instrumentation traces contained in the instrumentation trace cache. All future attempts to match an instrumentation trace will operate only against those traces received after this method was invoked.

Implementation

```
<<rein configuration>>=
method clearInstrTraceCache {} {
    my variable instrReceived
    while {[$instrReceived size] != 0} {
        $instrReceived get
    }
}
```

Format Instrumentation Traces

reinobj `formatInstrTraces` *?begin? ?finish?*

`begin`
An optional number giving the starting trace to include in the output. If missing, then `begin` is taken as 1, the first trace accumulated.

`finish`
An optional number giving the last trace to include in the output. If missing, then `finish` is taken as the last trace accumulated.

The `formatInstrTraces` method returns a string containing a human readable print out of the received instrumentation traces.

Implementation

```
<<rein configuration>>=
method formatInstrTraces {{begin 1} {finish end}} {
    set listing [list]
    set instrs [my RestrictTraces $begin $finish Instrumentation]

    relation foreach instr $instrs -ascending TraceId {
        relation assign $instr
        lappend listing [format "% 3u %s: %s" $TraceId $Time $Message]
    }

    return [join $listing "\n"]
}
```

Discard Instrumentation Traces

reinobj `discardInstrTraces`

The `discardInstrTraces` method deletes the set of accumulated instrumentation traces.

Implementation

```
<<rein configuration>>=
method discardInstrTraces {} {
    my variable instrId
```

```
    set instrId 0
    relvar eval {
        relvar set Instrumentation [relation empty [relvar set Instrumentation]]
    }
    return
}
```

Instrumentation Notification

reinobj `instrNotify` *?prefix?*

prefix
A command prefix.

The `instrNotify` method returns command prefix for a command that is executed whenever an instrumentation trace arrives from a test harness. The `prefix` argument is optional. If present, then the argument is interpreted as a command prefix and will be invoked with an additional argument as: `prefix` *instr* where *instr* is a dictionary with the instrumentation information. If missing, then no changes are made to the notification command. In both cases, the current value of the notification command prefix is returned. Setting the *prefix* to the empty string, disables instrumenation notification.

Implementation

```
<<rein configuration>>=
method instrNotify {args} {
    my variable instrPrefix
    if {[llength $args] == 1} {
        set instrPrefix [lindex $args 0]
    } elseif {[llength $args] > 1} {
        my Throw "wrong number of arguments"
    }
    return $instrPrefix
}
```

We must also provide a command that the event handler can use to notify when instrumenation arrived.

```
<<rein configuration>>=
method NotifyInstrArrived {instr} {
    my variable instrPrefix
    if {$instrPrefix eq {}} {
        return
    }

    try {
        eval [linsert $instrPrefix end $instr]
    } on error {result} {
        log::error "error notifying instrumentation arrival: \"$result\""
        set instrPrefix {}
    }
}
```

Instrumentation Trace Format

`mecate instrTraceFormat` *trace*

`trace`
An instrumentation trace in dictionary form as returned from the test harness.

The `instrTraceFormat` procedure returns a string containing a human readable version of the *trace*.

Implementation

```
<<mecate commands>>=
proc ::mecate::instrTraceFormat {trace} {
    dict with trace {
        return [format "%s: Instr: %s" $time $message]
    }
}
```

Fatal Error Traces

The `micca` run-time code detects several fatal errors. These errors can range from exhaustion of resources to a *can't happen* state transition. By default, the `micca` run-time invokes the `abort` function from the "C" standard library. Most implementations of `abort` cause the process to terminate and dump core for debugging.

When running as a test harness, terminating the process is not desirable. We would like to be able to examine the state of things after the fatal error has happened. So, `bosal` generates code to install its own fatal error handler and this handler allows the test harness communication socket to remain open. The fatal error handle also emits a fatal error message with the details. Note that the fatal error messages are sent asynchronously to any command / response sequence and cannot be tuned off.

Wait for Fatal Trace

reinobj `waitForFatalTrace` *?field1 pattern1 field2 pattern2?*

`fieldN`
The name of a field of the fatal error trace. Valid field names are:

`time`
A timestamp of when the trace was generated.

`message`
The text of the fatal error trace message.

`patternN`
The corresponding pattern to match for the given key. Patterns may be of any form accepted by the `string match` command.

The `waitForFatalTrace` method suspends execution until a fatal error trace matches the given parameters or a timeout occurs.

Implementation

```
<<rein configuration>>=
method waitForFatalTrace {args} {
    my variable fatalReceived
    my variable fatalSyncVar

    set expect $args
    while {true} {
```

```
        while {[$fatalReceived size] != 0} {
            set actual [$fatalReceived get]

            if {[my MatchTrace $actual $expect]} {
                return $actual
            } else {
                log::debug "discarding trace, \"$actual\":\
                        failed to match, \"$expect\""
            }
        }

        set resolvedVarName [my varname fatalSyncVar]
        set timer [expr {$timeout > 0 ?\
            [after $timeout [list set $resolvedVarName TIMEOUT]] : {}}]
        vwait $resolvedVarName

        if {$fatalSyncVar eq "TIMEOUT"} {
            my Throw "timed out on receiving event traces"
        } else {
            if {$timer ne {}} {
                after cancel $timer
            }
        }
    }
}
```

Tests

```
<<exported tests>>=
test fatal-1.0 {
    Cause a can't happen transition and await the trace
} -setup {
} -cleanup {
} -body {
    catch {::apptest signal bookstore ShippingClerk fs booksSelected}
    set err [::apptest waitForFatalTrace message *]
    return [dict get $err message]
} -result {*CH*} -match glob
```

Clear the Fatal Trace Cache

reinobj `clearFatalTraceCache`

The `clearFatalTraceCache` method deletes any accumulated fatal error traces contained in the fatal error trace cache. All future attempts to match an fatal error trace will operate only against those traces received after this method was invoked.

Implementation

```
<<rein configuration>>=
method clearFatalTraceCache {} {
    my variable fatalReceived
    while {[$fatalReceived size] != 0} {
        $fatalReceived get
    }
}
```

Format Fatal Traces

reinobj `formatFatalTraces` *?begin? ?finish?*

`begin`
An optional number giving the starting trace to include in the output. If missing, then `begin` is taken as 1, the first trace accumulated.

`finish`
An optional number giving the last trace to include in the output. If missing, then `finish` is taken as the last trace accumulated.

The `formatFatalTraces` method returns a string containing a human readable print out of the received traces.

Implementation

```
<<rein configuration>>=
method formatFatalTraces {{begin 1} {finish end}} {
    set listing [list]
    set fatals [my RestrictTraces $begin $finish Fatal]

    relation foreach fatal $fatals -ascending TraceId {
        relation assign $fatal
        lappend listing [format "% 3u %s: %s" $TraceId $Time $Message]
    }

    return [join $listing "\n"]
}
```

Discard Fatal Traces

reinobj `discardFatalTraces`

The `discardFatalTraces` method deletes the set of accumulated fatal error traces.

Implementation

```
<<rein configuration>>=
method discardFatalTraces {} {
    variable fatalId
    set fatalId 0
    relvar eval {
        relvar set Fatal [relation empty [relvar set Fatal]]
    }
    return
}
```

Fatal Notification

reinobj `fatalNotify` *?prefix?*

`prefix`
A command prefix.

The `fatalNotify` method returns command prefix for a command that is executed whenever a fatal error trace arrives from a test harness. The `prefix` argument is optional. If present, then the argument is interpreted as a command prefix and will be invoked with an additional argument as: `prefix` *fatal* where *fatal* is a dictionary with the fatal error information. If missing, then no changes are made to the notification command. In both cases, the current value of the notification command prefix is returned. Setting the *prefix* to the empty string, disables fatalumenation notification.

Implementation

```
<<rein configuration>>=
method fatalNotify {args} {
    my variable fatalPrefix
    if {[llength $args] == 1} {
        set fatalPrefix [lindex $args 0]
    } elseif {[llength $args] > 1} {
        my Throw "wrong number of arguments"
    }
    return $fatalPrefix
}
```

We must also provide a command that the event handler can use to notify when a fatal error traces arrived.

```
<<rein configuration>>=
method NotifyFatalArrived {fatal} {
    my variable fatalPrefix
    if {$fatalPrefix eq {}} {
        return
    }

    try {
        eval [linsert $fatalPrefix end $fatal]
    } on error {result} {
        log::error "error notifying fatal error arrival: \"$result\""
        set fatalPrefix {}
    }
}
```

Fatal Trace Format

`mecate fatalTraceFormat` *trace*

`trace`
An fatal error trace in dictionary form as returned from the test harness.

The `fatalTraceFormat` procedure returns a string containing a human readable version of the *trace*.

Implementation

```
<<mecate commands>>=
proc ::mecate::fatalTraceFormat {trace} {
    dict with trace {
        return [format "%s: Fatal: %s" $time $message]
    }
}
```

Save Traces

reinobj `saveTraces` *filename ?ral | sqlite?*

`filename`
The name of the file to which trace data is stored.

`ral | sqlite`
A string to designate the type of file to which trace data is stored. If `ral` is given or the option is missing, trace data is stored in the native serialization format of TclRAL. If `sqlite` is given, then trace data is stored in SQLite database.

The `saveTraces` method stores any accumulated event, instrumentation and fatal traces to a file.

Implementation

```
<<rein configuration>>=
method saveTraces {file {how ral}} {
    switch -exact -- $how {
        ral {
            serializeToFile $file [self namespace]::*
        }
        sqlite {
            storeToSQLite $file [self namespace]::*
        }
        default {
            my Throw "unknown trace file type, \"$how\""
        }
    }
    return
}
```

Handling Harness I/O

In this section, we show the methods of the `rein` class used to deal with communicating to the test harness program. Since we can enable event and instrument traces, and since those traces can occur asynchronously to any command / response sequence with the test harness, this design runs all the communications in a non-blocking asynchronous manner. This is easily accomplished in Tcl by setting the channel to be non-blocking and use a file event to handle arriving input. The format of the responses coming from a test harness have, as their first word, the type of the response. This lets us select the specific code to handle that particular type of input.

The asynchronous nature of the I/O to the test harness does cause some jumping around in the execution flow of the code. That makes things more difficult to explain. Consider, for example, dealing with a command / response to / from the test harness. After the command is send, the script waits on a variable, entering the Tcl event loop. When input arrives, a callback happens (at the global interpreter level) to the script registered on the communication channel. That script then reads the input from the channel, decides what to do and then writes a value to the variable upon which the script had previously waited. The callback finishes and control then returns from waiting on the variable to the next command in the script. This is generally how callbacks happen in Tcl, so there is nothing special about the way the `rein` class operates. It just requires a little tracking round to see the execution flow.

Sending commands to a test harness is simple. The protocol is ASCII record oriented and the core `puts` command does what is needed.

```
<<rein configuration>>=
method Puts {cmd} {
    log::debug "harness command: \"$cmd\""
    puts $connection $cmd
```

```
}
```

Handling input is more complicated. The `connect` method registers a file readable event on the communication socket so that the `HandleHarnessInput` method is invoked when input has arrived. Handling the input is a two step process:

1. Obtain a valid response
2. Invoke a method to handle the response based on its type.

```
<<rein configuration>>=
method HandleHarnessInput {} {
    <<HandleHarnessInput: get input line>>
    <<HandleHarnessInput: handle different response types>>
}
```

The only complication with obtaining a valid response deals with the possibility of receiving a partial response which might contain a newline character. We use the `chan gets` procedure to read in a line of input. All valid responses from a test harness are in the form of a proper Tcl list (not my accident). So we let the `string is list` command determine if we have obtained a line that could be a response. We continue to read input until we do get a proper Tcl list and then consider that as a test harness response.

```
<<HandleHarnessInput: get input line>>=
set line {}
for {set gotline false} {!$gotline} {set gotline [string is list $line]} {
    set llen [chan gets $connection input]
    if {$llen == -1} {
        if {[chan eof $connection]} {
            log::warn "EOF on harness input -- closing \"$connection\""
            chan close $connection
            unset -nocomplain connection
            return
        } elseif {[chan blocked $connection]} {
            log::notice "partial line received"
        }
    } else {
        append line $input
    }
}
```

Once we have a valid Tcl list, then we look at the first word of the list to obtain the type of the response. Actions on the response are delegated to separate methods based on the response type.

```
<<HandleHarnessInput: handle different response types>>=
if {[string length $line] != 0} {
    log::debug "harness response: \"$line\""

    lassign $line resp_type resp_value

    switch -exact -- $resp_type {
        cmd {
            my HandleCmdResponse $resp_value
        }
        trace {
            my HandleTraceResponse $resp_value
        }
        fatal {
            my HandleFatalResponse $resp_value
        }
        instr {
            my HandleInstrResponse $resp_value
        }
```

```
        default {
            my Throw "unknown response type, \"$resp_value\""
        }
    }
}
```

Handling Responses

The `rein` class methods provide for suspending execution to await responses from the test harness. Those responses will arrive via a file read event, so we must enter the Tcl event loop to receive the responses. The core command to accomplish that is `vwait`.

Code to handle the various types of responses follows a similar pattern. A distinct variable is used for wait for each type of response. The response handling methods for each response type write to the variable to effect the synchronization. Of course there is some different processing that also goes on for each distinct response type.

For commands, it is only necessary to write to the variable being used to synchronize the command responses.

```
<<rein configuration>>=
method HandleCmdResponse {respValue} {
    set [my varname cmdSyncVar] $respValue
    return
}
```

Event traces are more complicated. Here we have two additional concerns.

1. We need to keep a queue of the traces that can be examined later should the client wish to wait for a particular trace. Traces can be emitted at almost any time, we can't just drop them if no one is waiting at that moment.
2. We want to save all the event traces that occur so we can save them at the end of a test session.

```
<<rein configuration>>=
method HandleTraceResponse {respValue} {
    my variable traceId
    incr traceId

    relvar eval {
        switch -exact -- [dict get $respValue type] {
            transition {
                relvar insert TransitionTrace [list\
                    TraceId     $traceId\
                    Current     [dict get $respValue currstate]\
                    New         [dict get $respValue newstate]\
                ]
            }
            polymorphic {
                relvar insert PolymorphiceTrace [list\
                    TraceId         $traceId\
                    Relationship    [dict get $respValue relationship]\
                    NewEvent        [dict get $respValue newevent]\
                    Subclass        [dict get $respValue subclass]\
                ]
            }
            creation {
                relvar insert CreationTrace [list\
                    TraceId     $traceId\
                ]
            }
            default {
                log::error "unknown state machine trace type,\
```

```
                        \"[dict get $respValue type]\""
                incr traceId -1
                return                                                  ; # ❶
            }
        }

        relvar insert Trace [list\
            TraceId     $traceId\
            Time        [dict get $respValue time]\
            Event       [dict get $respValue event]\
            Source      [dict get $respValue source]\
            Target      [dict get $respValue target]\
        ]
    }

    my variable tracesReceived
    $tracesReceived put $respValue
    set [my varname traceSyncVar] TRACE

    my NotifyTraceArrived $respValue

    return
}
```

❶ Unrecognized trace types are discarded. They probably wouldn't match, but we avoid bad data anyway. We have to patch up the `traceId` value and make sure we don't insert any extraneous traces into the relvar storage or the cache.

The event dispatch traces are saved away in TclRAL relvars. For event traces, we need some schema to account for the various subtypes of trace data that arise from the different types of events.

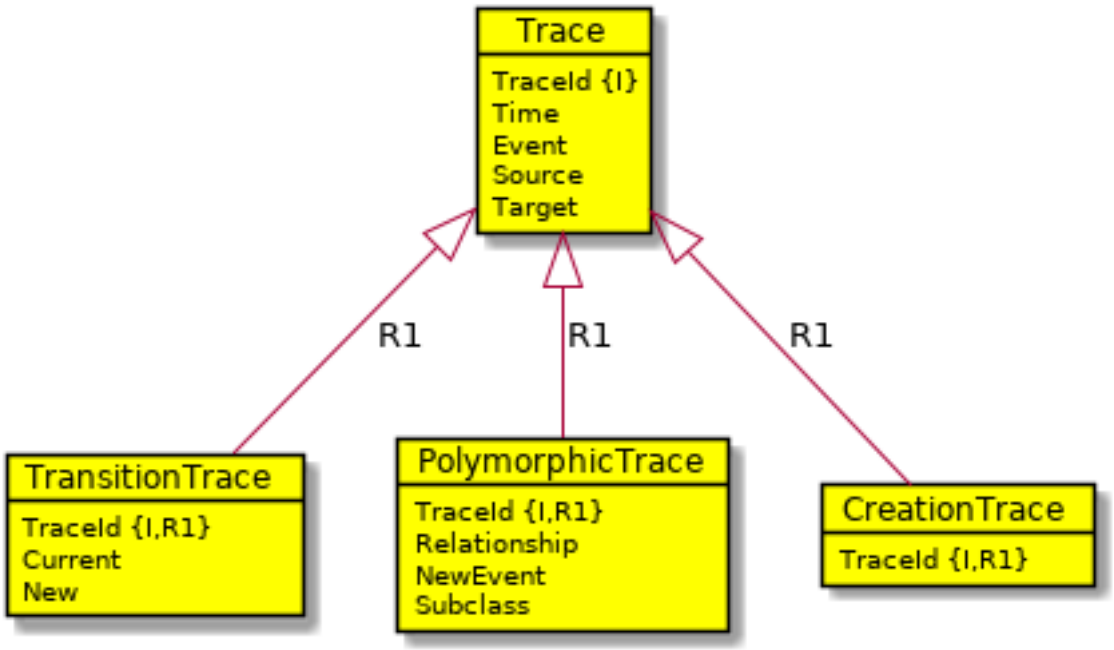

Figure 1: Class Diagram for State Machine Traces

The schema is represented as TclRAL `relvar create` commands.

```
<<rein data structures>>=
relvar create Trace {
    TraceId     int
    Time        string
    Event       string
    Source      string
    Target      string
} TraceId

relvar create TransitionTrace {
    TraceId     int
    Current     string
    New         string
} TraceId

relvar create PolymorphicTrace {
    TraceId         int
    Relationship    string
    NewEvent        string
    Subclass        string
} TraceId

relvar create CreationTrace {
    TraceId         int
} TraceId

relvar partition R1 Trace TraceId\
    TransitionTrace TraceId\
    PolymorphicTrace TraceId\
    CreationTrace TraceId
```

Handling instrumentation traces follows a similar pattern. Like event traces, we must keep a queue of instrumentation traces that can be searched later. We also store away the traces so they can all be saved later.

```
<<rein configuration>>=
method HandleInstrResponse {respValue} {
    variable instrReceived
    $instrReceived put $respValue

    variable instrId
    incr instrId
    relvar insert Instrumentation [list\
        TraceId     $instrId\
        Time        [dict get $respValue time]\
        Message     [dict get $respValue message]\
    ]

    set [my varname instrSyncVar] INSTR

    my NotifyInstrArrived $respValue

    return
}
```

The storage for instrumentation traces is much simpler. A single relvar will suffice.

```
<<rein data structures>>=
relvar create Instrumentation {
    TraceId     int
    Time        string
    Message     string
```

```
} TraceId
```

Finally, the fatal error traces follow the same pattern. A queue is kept as a cache for later searching and the fatal messages are kept for later storage. Note however, that fatal error messages do have one twist to them. After a fatal response is received, there will not be any other responses forthcoming from the test harness until another command is sent. Since we don't know which variable might be used for waiting, a fatal response must wake up any waiting script. This is accomplished by simply writing to each of the three synchronization variables.

```
<<rein configuration>>=
method HandleFatalResponse {respValue} {
    my variable fatalReceived
    $fatalReceived put $respValue

    variable fatalId
    incr fatalId
    relvar insert Fatal [list\
        TraceId     $fatalId\
        Time        [dict get $respValue time]\
        Message     [dict get $respValue message]\
    ]

    log::error "received fatal error response, \"$respValue\""
    set [my varname fatalSyncVar] FATAL
    set [my varname cmdSyncVar] FATAL
    set [my varname traceSyncVar] FATAL

    my NotifyFatalArrived $respValue

    return
}
```

Again, a simple relvar is all that is necessary to save fatal error responses.

```
<<rein data structures>>=
relvar create Fatal {
    TraceId     int
    Time        string
    Message     string
} TraceId
```

Command

Finally, we have enough background to show the `Command` method. The `Command` method sends the harness program a `harness` command and waits for the response. All the methods that represent commands to the test harness are just thin veneers, using the `Command` method to do most of the work.

Implementation

```
<<rein configuration>>=
method Command {args} {
    set resolvedVarName [my varname cmdSyncVar]
    set timer [expr {$timeout > 0 ?\
        [after $timeout [list set $resolvedVarName TIMEOUT]] : {}}]

    my Puts $args                                   ; # ❶

    vwait $resolvedVarName                          ; # ❷

    set cmdName [lindex $args 0]
    my variable cmdSyncVar
```

```
    if {$cmdSyncVar eq "TIMEOUT"} {
        my Throw "timeout for command, \"$cmdName\""
    } elseif {$cmdSyncVar eq "FATAL"} {
        my Throw "fatal run-time error while executing, \"$cmdName\""
    } else {
        if {$timer ne {}} {
            after cancel $timer
        }
    }

    if {[dict get $cmdSyncVar name] ne $cmdName} {
        my Throw "expected response for command,\
                \"$cmdName\",\
                got, \"[dict get $cmdSyncVar name]\""
    }

    set result [dict get $cmdSyncVar result]        ; # ❸
    if {[dict get $cmdSyncVar status] eq "error"} {
        my Throw $result
    }

    return $result
}
```

❶ Send the command to the test harness.

❷ Wait for the response.

❸ Note we only return the `result` field of the response as the return value of the command.

Finally, we have a small utility for error handling that insures that a reasonable error code is available. This eases programatic error handling.

```
<<rein configuration>>=
method Throw {message} {
    tailcall uplevel 1 [list throw [list {*}[self caller] $message] $message]
}
```

Bookstore Execution Scenarios

In this section, a set of test cases is presented that execute the Mellor/Balcer simple bookstore model. This model was translated to micca and that translation can be found here. The execution scenarios in that translation were accomplished by directly coding the interactions with the domain as part of the `main()` function. Here we have removed that `main` function and create a test harness using `bosal` which supplied only a simple `main`. The same scenarios are then executed using `mecate` and the `bosal` generated test harness. The results are the same, however, now the execution scenarios can be changed or extended without having to modify the `main` function or any other parts of the test harness. This represents the added flexibility that a test harness provides.

```
<<scenario tests>>=
test scenario-0.1 {
    Start session with bookstore domain
} -setup {
} -cleanup {
} -body {
    ::apptest disconnect

    ::apptest clearEventTraceCache
```

```
    ::apptest discardEventTraces

    ::apptest clearInstrTraceCache
    ::apptest discardInstrTraces

    ::apptest clearFatalTraceCache
    ::apptest discardFatalTraces

    after 500
    ::apptest start ./bstore_harness
    ::apptest connect

    ::apptest traceNotify [namespace code {logTrace info}]
    ::apptest instrNotify [namespace code {logInstr info}]
    ::apptest fatalNotify [namespace code {logFatal error}]

    ::apptest instr on
    ::apptest trace on
} -result {on}
```

```
<<scenario tests>>=
test scenario-1.0 {
    Create a shopping cart asynchronously
} -setup {
    set prodattr [::apptest read bookstore Product mbxuml productID]
    set prodid [dict get $prodattr productID]
} -cleanup {
} -body {
    set cartid [::apptest createasync bookstore ShoppingCart\
            startCart $prodid 1]
    ::apptest waitForEventTrace\
            type transition\
            target ShoppingCart.$cartid\
            newstate AddingSelectiontoOrder
    return $cartid
} -result {0}
```

```
<<scenario tests>>=
test scenario-1.1 {
    Cancel order by changing quantity to 0.
} -setup {
    set prodinsts [::apptest query instances bookstore Product]
    set mbxumlinst [dict get $prodinsts named mbxuml]
} -cleanup {
} -body {
    ::apptest domainop bookstore changeSelectionQuantity\
        $cartid $mbxumlinst 0
    set trace [::apptest waitForEventTrace type transition event cancel\
            target ShoppingCart.$cartid]
    return [dict get $trace newstate]
} -result {CancelingEntireOrder}
```

```
<<scenario tests>>=
test scenario-2.0 {
    Create a new shopping cart
} -setup {
    set prodattr [::apptest read bookstore Product mbxuml productID]
    set prodid [dict get $prodattr productID]
} -cleanup {
} -body {
    set cartid [::apptest createasync bookstore ShoppingCart\
```

```
            startCart $prodid 1]
    ::apptest waitForEventTrace\
            type transition\
            target ShoppingCart.$cartid\
            newstate AddingSelectiontoOrder
    return $cartid
} -result {1}
```

```
<<scenario tests>>=
test scenario-2.1 {
    Buy three copies of the book
} -setup {
    set prodinsts [::apptest query instances bookstore Product]
    set mbxumlinst [dict get $prodinsts named mbxuml]
} -cleanup {
} -body {
    ::apptest domainop bookstore changeSelectionQuantity\
        $cartid $mbxumlinst 3
    set trace [::apptest waitForEventTrace type transition event changeQuantity\
            target ProductSelection.1]
    return [dict get $trace newstate]
} -result {ChangingQuantity}
```

```
<<scenario tests>>=
test scenario-2.2 {
    Add some music to the shopping cart
} -setup {
    set prodattr [::apptest read bookstore Product cmc productID]
    set prodid [dict get $prodattr productID]
} -cleanup {
} -body {
    ::apptest signal bookstore ShoppingCart $cartid addSelection $prodid 2
    ::apptest waitForEventTrace\
            type transition\
            target ShoppingCart.$cartid\
            newstate AddingSelectiontoOrder
    return $cartid
} -result {1}
```

```
<<scenario tests>>=
test scenario-2.3 {
    Checkout to buy selections
} -setup {
} -cleanup {
} -body {
    ::apptest signal bookstore ShoppingCart $cartid checkOut\
            "5555 6666 7777 8888"\
            "100 E Main St., Anywhere, CO 88888"\
            "9/11/2010"\
            "Fred Smith"\
            "fsmith@gmail.com"\
            "Fred Smith"\
            "222-555-1212"\
            "100 E Main St., Anywhere, CO 88888"
    set event [::apptest waitForEventTrace\
            type transition\
            source Warehouse.acme\
            target ShippingClerk.fs\
            newstate SelectingBooks]
    return [dict get $event newstate]
} -result {SelectingBooks}
```

```
<<scenario tests>>=
test scenario-3.0 {
    Wait for order to be packed
} -setup {
} -cleanup {
} -body {
    set event [::apptest waitForEventTrace\
            type transition\
            target Warehouse.acme\
            currstate WaitingforaShipment\
            newstate IG]
    return [dict get $event newstate]
} -result {IG}
```

```
<<scenario tests>>=
test scenario-4.0 {
    Wait for order to be delivered
} -setup {
} -cleanup {
} -body {
    set event [::apptest waitForEventTrace\
            type transition\
            target Order.0\
            newstate DeliveredtoCustomer]
    return [dict get $event newstate]
} -result {DeliveredtoCustomer}
```

```
<<scenario tests>>=
test scenario-5.0 {
    Save traces
} -setup {
} -cleanup {
} -body {
    ::apptest saveTraces scenarios.ral
    ::apptest saveTraces scenarios.sqlite sqlite
} -result {}
```

Tests

```
<<scenario tests>>=
test scenario-5.0.1 {
    Create sequence diagram
} -setup {
} -cleanup {
} -body {
    ::apptest seqDiagToFile scenario-seq.txt
} -result {}
```

```
<<scenario tests>>=
test scenario-5.1 {
    Print traces
} -setup {
} -cleanup {
} -body {
    puts "***"
    puts "*** State Machine Event Traces"
    puts "***"
    puts [::apptest formatEventTraces]

    puts "***"
```

```
    puts "*** Instrumentation Traces"
    puts "***"
    puts [::apptest formatInstrTraces]

    puts "***"
    puts "*** Fatal Error Traces"
    puts "***"
    puts [::apptest formatFatalTraces]
} -result {}
```

```
<<scenario tests>>=
test scenario-5.2 {
    Discard event traces
} -setup {
} -cleanup {
} -body {
    ::apptest discardEventTraces
    ::apptest formatEventTraces
} -result {}
```

```
<<scenario tests>>=
test scenario-5.3 {
    Discard instrumentation traces
} -setup {
} -cleanup {
} -body {
    ::apptest discardInstrTraces
    ::apptest formatInstrTraces
} -result {}
```

```
<<scenario tests>>=
test scenario-5.4 {
    Discard fatal error traces
} -setup {
} -cleanup {
} -body {
    ::apptest discardFatalTraces
    ::apptest formatFatalTraces
} -result {}
```

The results of running the scenarios can be seen in the following UML sequence diagram.

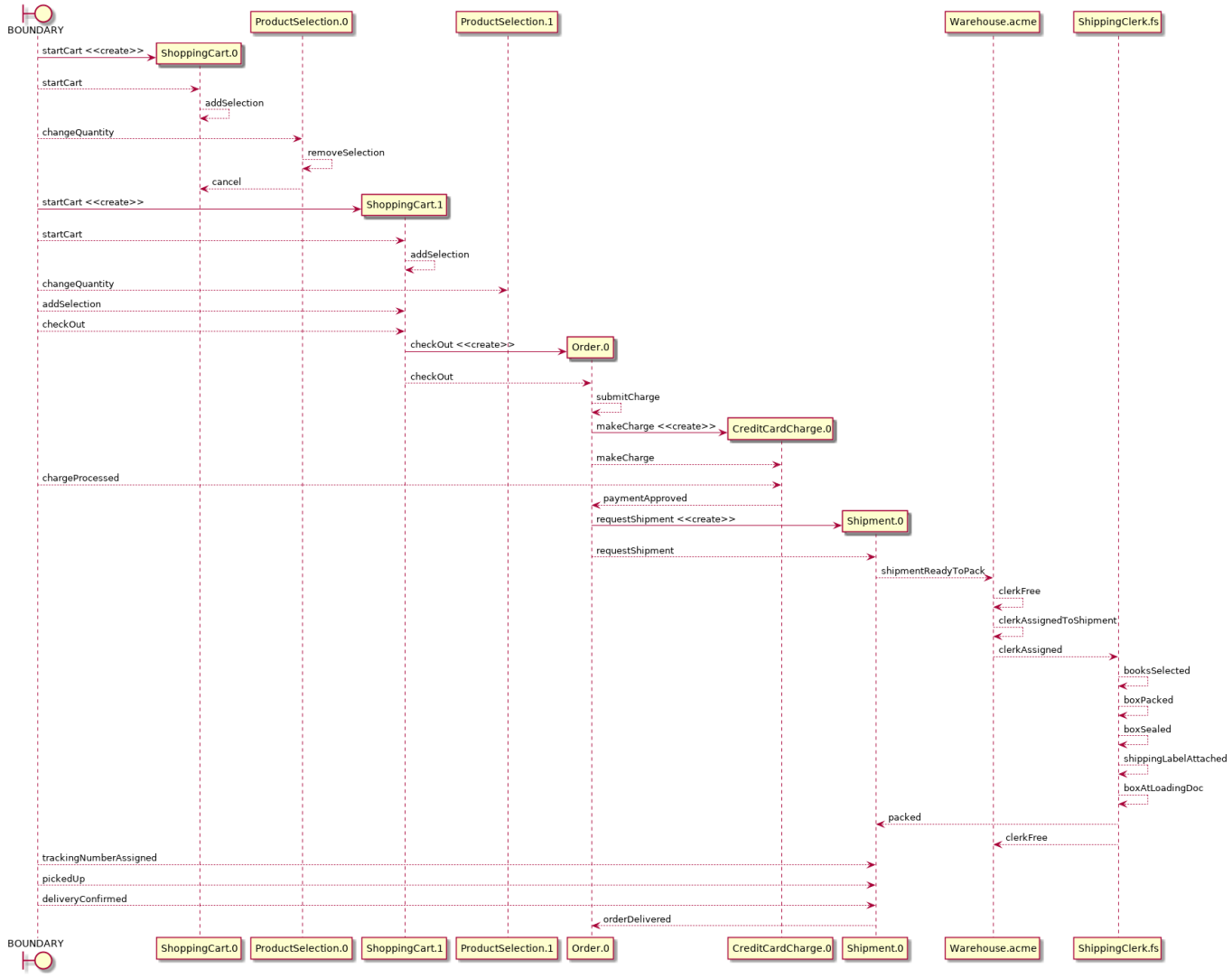

Figure 2: Sequence Diagram of Scenario State Machine Traces

Code Organization

This section shows how the literate program chunks are organized into source files. There are several literate program **roots** in this document and each is discussed in the sections below.

Source Code

The source code for the `mecate` package is shown below.

```
<<mecate.tcl>>=
<<edit warning>>
#
<<copyright info>>
#

package require Tcl 8.6
package require logger
package require logger::utils
```

```
package require logger::appender
package require ral
package require ralutil
package require oo::util
package require struct::queue

namespace eval ::mecate {
    namespace export rein
    namespace export eventTraceFormat
    namespace export instrTraceFormat
    namespace export fatalTraceFormat
    namespace ensemble create

    set logger [::logger::init mecate]
    set appenderType [expr {[dict exist [fconfigure stdout] -mode] ?\
            "colorConsole" : "console"}]
    ::logger::utils::applyAppender -appender $appenderType -serviceCmd $logger\
            -appenderArgs {-conversionPattern {\[%c\] \[%p\] '%m'}}
    ::logger::import -all -force -namespace log mecate

    variable version 1.1
}

<<mecate commands>>

package provide mecate $::mecate::version
```

Mecate Unit Tests

This unit tests cases included in this document are placed in a single test file.

```
<<mecate.test>>=
#!/usr/bin/env tclsh
#
<<edit warning>>
#
<<copyright info>>
#

package require Tcl 8.6
package require cmdline
package require logger
package require logger::utils
package require logger::appender
package require tcltest

source ../code/tcl/mecate.tcl

# Add custom arguments here.
set optlist {
    {level.arg warn {Log debug level}}
}
array set options [::cmdline::getKnownOptions argv $optlist]

::mecate::log::setlevel $options(level)

tcltest::configure {*}$argv

namespace eval ::mecate::test {
    set logger [::logger::init mecate::test]
```

```
    set appenderType [expr {[dict exist [fconfigure stdout] -mode] ?\
            "colorConsole" : "console"}]
    ::logger::utils::applyAppender -appender $appenderType -serviceCmd $logger\
            -appenderArgs {-conversionPattern {\[%c\] \[%p\] '%m'}}
    ::logger::import -all -force -namespace log mecate::test

    log::info "testing mecate version: [package require mecate]"

    namespace import ::tcltest::*
    namespace import ::ral::*
    namespace import ::ralutil::*

    <<test utilities>>
    <<constructor tests>>
    <<unexported tests>>
    <<exported tests>>
    <<scenario tests>>

    ::apptest destroy

    cleanupTests
}
```

```
<<test utilities>>=
# Utility proc to execute unexported methods.
proc objeval {obj args} {
    namespace eval [info object namespace $obj] {*}$args
}
```

```
<<test utilities>>=
# Utility proc log a formated event trace
proc logTrace {level trace} {
    log::$level [::mecate eventTraceFormat $trace]
}
proc logInstr {level trace} {
    log::$level [::mecate instrTraceFormat $trace]
}
proc logFatal {level trace} {
    log::$level [::mecate fatalTraceFormat $trace]
}
```

Package Index

We also provide a root chunk to extract a package index file.

```
<<pkgIndex.tcl>>=
package ifneeded mecate 1.1 [list source [file join $dir mecate.tcl]]
```

Template

The following is a template that can be used to seed a unit test file that uses `mecate`.

```
<<template-mecate.test>>=
#  *++
# MODULE:
#       unit test template
# ABSTRACT:
#
```

```
#  *--
package require Tcl 8.6
package require cmdline
package require logger
package require logger::utils
package require logger::appender
package require tcltest
package require mecate

set optlist {
    {level.arg warn {Log debug level}}
    {program.arg {} {Name of test harness program}}
    {nostart {Don't start test harness program automatically}}
}
array set options [::cmdline::getKnownOptions argv $optlist]

::logger::setlevel $options(level)

tcltest::configure {*}$argv

namespace eval ::$options(program)::test {
    set logger [::logger::init $::options(program)::test]
    set appenderType [expr {[dict exist [fconfigure stdout] -mode] ?\
            "colorConsole" : "console"}]
    ::logger::utils::applyAppender -appender $appenderType -serviceCmd $logger\
            -appenderArgs {-conversionPattern {\[%c\] \[%p\] '%m'}}
    ::logger::import -all -force -namespace log $::options(program)::test

    if {$options(program) eq {}} {
        my Throw "no bosal test harness program name was supplied:\
                use \"-program <name>\" option"
    }

    namespace import ::tcltest::*

    mecate rein create ::apptest

    if {!$::options(nostart)} {
        apptest start
    }

    # Here you can turn on any traces that are needed.
    #
    # For example:
    #
    # ::apptest trace on
    # ::apptest instr on
    # ::apptest eloop halt  ; # Start halted so we can single step events.


    # Place your text cases here.
    #
    # For example:

    test init-1.0 {
        Initialize the domain
    } -setup {
    } -cleanup {
    } -body {
        ::apptest domainop MyDomain init
    } -result {}
```

```
    apptest destroy

    cleanupTests
}

if 0 {
    # This is just a template for additional tests.
    # You can paste it into your test set to save some typing.

    test X-1.0 {
    } -setup {
    } -cleanup {
    } -body {
    } -result {}
}
```

Copyright Information

```
<<copyright info>>=
# This software is copyrighted 2018 by G. Andrew Mangogna.
# The following terms apply to all files associated with the software unless
# explicitly disclaimed in individual files.
#
# The authors hereby grant permission to use, copy, modify, distribute,
# and license this software and its documentation for any purpose, provided
# that existing copyright notices are retained in all copies and that this
# notice is included verbatim in any distributions. No written agreement,
# license, or royalty fee is required for any of the authorized uses.
# Modifications to this software may be copyrighted by their authors and
# need not follow the licensing terms described here, provided that the
# new terms are clearly indicated on the first page of each file where
# they apply.
#
# IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR
# DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING
# OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES
# THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF
# SUCH DAMAGE.
#
# THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,
# INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.  THIS SOFTWARE
# IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE
# NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS,
# OR MODIFICATIONS.
#
# GOVERNMENT USE: If you are acquiring this software on behalf of the
# U.S. government, the Government shall have only "Restricted Rights"
# in the software and related documentation as defined in the Federal
# Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2).  If you
# are acquiring the software on behalf of the Department of Defense,
# the software shall be classified as "Commercial Computer Software"
# and the Government shall have only "Restricted Rights" as defined in
# Clause 252.227-7013 (c) (1) of DFARs.  Notwithstanding the foregoing,
# the authors grant the U.S. Government and others acting in its behalf
# permission to use and distribute the software in accordance with the
# terms specified in this license.
```

```
<<edit warning>>=
# DO NOT EDIT THIS FILE!
```

```
# THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
```

Literate Programming

The source for this document conforms to asciidoc syntax. This document is also a literate program. The source code for the implementation is included directly in the document source and the build process extracts the source that is then given to the Tcl interpreter. This process is known as *tangle*ing. The program, `atangle`, is available to extract source code from the document source and the `asciidoc` tool chain can be used to produce a variety of different output formats, although PDF is the intended choice.

The goal of a literate program is to explain the logic of the program in an order and fashion that facilitates human understanding of the program and then *tangle* the document source to obtain the code in an order suitable for a compiler or interpreter. Briefly, code is extracted from the literate source by defining a series of *chunks* that contain the source. A chunk is *defined* by including its name as:

```
<<chunk name>>=
```

The trailing = sign denotes a definition. A chunk definition ends at the end of the source block or at the beginning of another chunk definition. A chunk may be *referenced* from within a chunk definition by using its name without the trailing = sign, as in:

```
<<chunk definition>>=
    <<chunk reference>>
```

Chunk names are arbitrary strings. Multiple definitions with the same name are simply concatenated in the order they are encountered. There are one or more *root chunks* which form the conceptual tree for the source files that are contained in the literate source. By convention, root chunks are named the same as the file name to which they will be tangled. Tangling is then the operation of starting at a root chunk and recursively substituting the definition for the chunk references that are encountered.

For readers that are not familiar with the literate style and who are adept at reading source code directly, the chunks definitions and reordering provided by the tangle operation can be a bit disconcerting at first. You can, of course, examine the tangled source output, but if you read the program as a document, you will have to trust that the author managed to arrange the chunk definitions and references in a manner so that the tangled output is acceptable to the Tcl interpreter.

Index