Daut

Version 2.0

Daut is programmed in Scala.

Monitoring Data Streams with Data Automata

Daut (Data automata) is an internal Scala DSL for writing event stream monitors. It supports a simple but yet interesting combination of state machines, temporal logic, and rule-based programming, all in one unified formalism, implemented in very few lines of code (341 lines not counting comments). The underlying concept is that at any point during monitoring there is an active set of states, the *"state soup"*. States can be added to this set by taking state to state transitions (target states are added), and can be removed from this soup by leaving states as a result of transitions. Each state in the soup can itself monitor the incoming event stream, and it can be used to record data (as in rule-based programming).

The specification language specifically supports:

- Automata, represented by states, parameterized with data (thereby the name Daut: Data automata).
- Temporal operators which generate states, resulting in more succinct specifications.
- Rule-based programming in that one can test for the presence of states.
- General purpose programming in Scala when the other specification features fall short.

The DSL is a simplification of the TraceContract https://github.com/havelund/tracecontract internal Scala DSL, which was used on the LADEE mission for checking command sequences against flight rules.

The general idea is to create a monitor as a class sub-classing the `Monitor` class, create an instance of it, and then feed it with events with the `verify(event: Event)` method, one by one, and in the case of a finite sequence of observations, finally calling the `end()` method on it. If `end()` is called, it will be determined whether there are any outstanding obligations that have not been satisfied (expected events that did not occur).

This can schematically be illustrated as follows:

```
import daut._

class MyMonitor extends Monitor[SomeType] {
  ...
}

object Main {
  def main(args: Array[String]) {
    val m = new MyMonitor()
    m.verify(event1)
    m.verify(event2)
    ...
    m.verify(eventN)
    m.end()
  }
}
```

In the following, we shall illustrate the API by going through a collection of examples.

Installation with SBT (assumes knowledge of SBT and Scala)

There are some options:

- The repository is an SBT project. To run it as an SBT project you must download SBT first, and follow the instructions there on how to run SBT projects.

- The main file is: src/main/scala/daut/Monitor.scala. This source can alternatively be incorporated into your own project, fitting your own way of working, and you are up and running. One file is simplicity.

- You may use an IDE, such as IntelliJ or Eclipse. IntelliJ can load the project in SBT mode.

Installation with jar file (command line use)

Download the jar files:

```
https://github.com/havelund/daut/tree/master/out/artifacts/daut_jar/daut.ja
https://github.com/havelund/daut/tree/master/out/artifacts/daut_jar/fastcsv
```

and compile and run examples as follows.

First, define a path to the jar files (here using bash export):

```
export DAUTMAIN=path/to/daut.jar
export CSV=path/to/fastcsv-1.0.1.jar
export DAUT=$CSV:$DAUTMAIN
```

The jar file is generated with Scala 2.13.3.

Now let's compile a program. Go to `src/test/scala/daut1_temporal`. Here is a file `Main.scala` with the following contents:

```
package daut1_temporal

import daut._

/**
 * Property AcquireRelease: A task acquiring a lock should eventually relea
 * can acquire a lock at a time.
 */

trait LockEvent
case class acquire(t:Int, x:Int) extends LockEvent
case class release(t:Int, x:Int) extends LockEvent

class AcquireRelease extends Monitor[LockEvent] {
  always {
    case acquire(t, x) =>
      hot {
        case acquire(_,`x`) => error
        case release(`t`,`x`) => ok
      }
  }
}

object Main {
  def main(args: Array[String]) {
    DautOptions.DEBUG = true
    val m = new AcquireRelease
    m.verify(acquire(1, 10))
    m.verify(release(1, 10))
    m.end()
  }
}
```

Compile as follows:

```
scalac -cp $DAUT Main.scala
```

This creates a directory with the name: `daut1_temporal` containing the compiled class files.

Now run the program as follows:

```
scala -cp .:$DAUT daut1_temporal.Main
```

This should generate the output:

```
===[acquire(1,10)]===

--- AcquireRelease:
[memory]
  hot
  always




===[release(1,10)]===

--- AcquireRelease:
[memory]
  always



Ending Daut trace evaluation for AcquireRelease
```

Some debugging information and no errors detected. The trace satisfies the specification.

Note: in Scala you indicate an "object" to run, in this case the object named `Main`. It must contain a `main` method. There can be several such objects in a file, `Main1`, `Main2`, each containing a `main` method. One picks which to execute in the above commmand.

Basic Example

Consider the monitoring of acquisition and release of locks by threads. We shall in other words monitor a sequence of events, where each event indicates either the acquisition of a lock by a thread, or the release of a lock by a thread. We can then formulate various policies about such acquisitions and releases as Daut monitors.

Events

Let us start by modeling the type `LockEvent` of events:

```
trait LockEvent
case class acquire(t:Int, x:Int) extends LockEvent // thread t acquires loc
case class release(t:Int, x:Int) extends LockEvent // thread t releases loc
```

The Property

We can then formulate our first property:

- *"A task acquiring a lock should eventually release it. At most one task can acquire a lock at a time"*.

The Monitor

This property is stated as the following monitor:

```
class AcquireRelease extends Monitor[LockEvent] {
  always {
    case acquire(t, x) =>
      hot {
        case acquire(_,`x`) => error
        case release(`t`,`x`) => ok
      }
  }
}
```

The monitor is formulated as a class `AcquireRelease` extending the `Monitor` class, instantiated with the type of events, `LockEvent`, that we want to monitor.

The body of the `AcquireRelease` monitor class shall read as follows: it is always the case (checked continuously) that if we observe an `acquire(t, x)` event, then we go into a so-called `hot` state (which must be left eventually), where we are waiting for one of two events to occur: either an:

```
acquire(_,`x`)
```

event, where the `x` is the same as was previously acquired (caused by the quotes around `x`), or a:

```
release(`t`,`x`)
```

event, where the thread `t` and the lock `t` are the same as in the original acquisition. In the first case the transition returns the `error` state, and in the second case the transition returns the `ok` state. An `ok` means that we are done monitoring that particular path in the monitor.

Applying the Monitor

We can now apply the monitor, as for example in the following main program:

```
object Main {
  def main(args: Array[String]) {
    val m = new AcquireRelease
    m.verify(acquire(1, 10))
    m.verify(release(1, 10))
    m.end()
  }
}
```

Note the call `m.end()` which terminates monitoring. A monitor does not need to be terminated by this call, and will not be if for example it concerns an ongoing online monitoring of an executing system. However, if this method is called, it will check that no monitor is in a `hot` state, as shown above. This is in effect how eventuallity properties are checked on finite traces.

The State Producing Functions

The body `always { ... }` above was perhaps a little bit of a mystery. In reality it is a call of a function with the signature:

```
type Transitions = PartialFunction[LockEvent, Set[state]]

def always(ts: Transitions): state
```

That is, `always` is a function that as argument takes a partial function from events to sets of states, and returns a state. Partial functions in Scala are represented as sequences of **case** statements of the form:

```
{
  case pattern_1 => code_1
  case pattern_2 => code_2
  ...
  case pattern_n => code_n
}
```

and this is exactly how we model transitions out of a state. The `always` function therefore returns a state with these transitions leaving it. In addition, since it is an always-state, it has a self loop back to itself, so taking a transition returns the state itself in addition to whatever the chosen transition produces as a set of states.

Daut offers a collection of such functions, each returning a state using the provided transition function according to its semantics. All states are final states except those produced by the functions `hot` and `next`, which are non-final states. Being in a non-final states results in an error when `end()` is called.

```
def always(ts: Transitions): state // always check the transitions
def watch(ts: Transitions): state  // watch until one of the transitions fi
def hot(ts: Transitions): state    // non-final, otherwise same meaning as
def next(ts: Transitions): state   // non-final, one of the transitions mus
def wnext(ts: Transitions): state  // one of the transtions must fire next,
```

Note: these functions actually return an object of a subclass (named `anonymous`) of class `state`, but that is not important here.

From States to Sets of States and Other Magic

You notice above that the state producing functions each takes a partial function of type `Transitions` as argument, that is, of type:

```
PartialFunction[LockEvent, Set[state]]
```

Such a partial function returns a **set** of states. In the above example, repeated here, however, we see that single states are returned, namely `hot {...}`, `error`, and `ok`.

```
always {
  case acquire(t, x) =>
    hot {
      case acquire(_,`x`) => error
      case release(`t`,`x`) => ok
    }
}
```

How does that work? - a state is not a set of states! An implicit function handles such cases, lifting (behind our back) any single state `S` to the set `Set(S)`, to make the program type check, and work:

```
implicit def convState2StateSet(state: state): Set[state] = Set(state)
```

Other such implicit functions exist, for example the following two, which respectively allow us to write code with side effects as the target of a transition, which will result in an `ok` state, or writing a Boolean expression, which will result in `ok` or `error` state depending on what the Boolean expression evaluates to:

```
implicit def convUnit2StateSet(u: Unit): Set[state] = Set(ok)
implicit def convBoolean2StateSet(b: Boolean): Set[state] = Set(if (b) ok e
```

How Did the Initial State Get Recorded?

You may wonder how the state procuced by the `always` function ends up being monitored. After all, it is just a call of a function that returns a state. It happens conveniently that any state created in a monitor before the first call of the `verify(event: Event)` method becomes part of the initial state of the monitor (a side effect of generating the state).

Naming the Intermediate State via a Function Call

We have already gotten a good sense of the temporal logic flavor of the Daut logic. However, Daut also supports state machine notation. Suppose we want to modify the above monitor to explicitly name the hot state where we have received a lock acquisition, but where we are waiting for a release, as one would do in a state machine. This can be done by simply defining the hot state as the body of a state returning function, and then call that function, as is done in the following:

```
class AcquireRelease extends Monitor[LockEvent] {
  always {
    case acquire(t, x) => acquired(t, x)
  }

  def acquired(t: Int, x: Int) : state =
    hot {
      case acquire(_,`x`) => error("lock acquired before released")
      case release(`t`,`x`) => ok
    }
}
```

There is no magic to this: the function call `acquired(t, x)` simply returns the hot state that we had previously inlined.

The previous temporal version is shorter and might be preferable. Naming states, however, can bring conceptual clarity to sub-concepts.

This "state machine" contains no loops. The next example introduces a looping state machine, with data.

Note that a text string can be passed to `error` as shown. This also holds for `ok`. These strings will be printed in various output.

Transitions Returning a Set of States

As mentioned already, a transition technically returns a set of states, each of which has to lead to success. Effectively, this set represents a *conjunction*: all these result states have to lead to succces. As an example of how this concept can be utilized for spceification of properties, consider the following property:

- *"A task acquiring a lock should eventually release it. A task can only acquire the same lock once"*.

This property can be formulated as the following monitor:

```
class AcquireRelease extends Monitor[LockEvent] {
  always {
    case acquire(t, x) =>
      Set(
        doRelease(t, x),
        dontAcquire(t,x)
      )
  }

  def doRelease(t: Int, x: Int) : state =
    hot {
      case release(`t`,`x`) => ok
    }

  def dontAcquire(t: Int, x: Int) : state =
    watch {
      case acquire(`t`,`x`) => error("lock acquired again by same task")
    }
}
```

The `AcquireRelease` monitor returns a set of states upon observing an `acquire` event. The first state (returned by `doRelease`) checks that the `release` event eventually follows. The second state (returned by `dontAcquire`) checks that another acquisition of that lock by that task does not occur.

Looping State Machines using Functions that

Return States

In this example, we shall illustrate a state machine with a loop, using functions to represent the individual states, which by the way in this case are parameterized with data, a feature not supported by text book state machines, including extended state machines.

The Events

We are monitoring a sequence of `start` and `stop` events, each carrying a task id as parameter:

```
trait TaskEvent
case class start(task: Int) extends TaskEvent
case class stop(task: Int) extends TaskEvent
```

The Property

The property we want to monitor is the following:

- *"Tasks should be executed (started and stopped) in increasing order according to task numbers, starting from task 0, with no other events in between, hence: start(0),stop(0),start(1),stop(1),start(2),stop(2),... A started task should eventually be stopped"*.

The Monitor

This following monitor verifies this property, and illustrates the use of next and weak next states.

```
class StartStop extends Monitor[TaskEvent] {
  def start(task: Int) : state =
    wnext {
      case start(`task`) => stop(task)
    }

  def stop(task: Int) : state =
    next {
      case stop(`task`) => start(task + 1)
    }

  start(0) // initial state
}
```

The Main Program

The following main program exercises the monitor.

```
object Main {
  def main(args: Array[String]) {
    val m = new StartStop
    m.PRINT = true
    m.verify(start(0))
    m.verify(stop(0))
    m.verify(start(1))
    m.verify(stop(1))
    m.verify(start(3)) // violates property
    m.verify(stop(3))
    m.end()
  }
}
```

States as Case Classes, towards Rule-Based Programming

Above we saw how state machines can be modeled using state-returning functions. There is an alternative way of modeling states, which becomes useful when we want to use techniques known from rule-based programming. In rule-based programming one can query whether a particular state is in the *state soup*, as a condition to taking a transition. This is particularly useful for modeling properties reasoning about the past (the past is stored as states). In order to do that, we need to make states objects (of case classes). This technique can be used as a general technique for modeling state machines, but is only strictly needed when quering states in this manner.

The Property

Let's go back to our lock acquisition and release scenario, and formulate the following property:

- *"A task acquiring a lock should eventually release it. At most one task can acquire a lock at a time. A task cannot release a lock it has not acquired."*.

It is the last requirement *"A task cannot release a lock it has not acquired"*, that is a past time property: if a task is released, it must have been acquired in the past, and not released since.

The Monitor

The monitor can be formulated as follows:

```
class AcquireRelease extends Monitor[LockEvent] {
  case class Locked(t: Int, x: Int) extends fact {
    hot {
      case acquire(_, `x`) => error
      case release(`t`, `x`) => ok
    }
  }

  always {
    case acquire(t, x) => Locked(t, x)
    case release(t, x) if !Locked(t, x) => error
  }
}
```

The monitor declares a **case** class `Locked`. An instance `Locked(t, x)` of this class is a state (the `Locked` class extends the `fact` trait, which itself extends the `state` trait) and is meant to represent the *fact* (rule-based terminology) that thread `t` has acquired the lock `x`. A state `Locked(t, x)` is created and added to the *state soup* upon the observation of an `acquire(t, x)` event in the always active `always` state. Note how the state itself this time checks whether it gets double acquired by some other thread (underscore pattern means that we don't care about the thread), and if released, resulting in the `ok` state, goes away.

Now, the same `always` state producing the `Locked(t, x)` state also contains the transition:

```
case release(t, x) if !Locked(t, x) => error
```

that tests for the occurrence of a `Locked(t, x)` state (fact) in the *state soup*, and if not present in case of a `release(t, x)` event yields an error. Here a little implicit function magic is occurring. The term `!Locked(t, x)` is the negation of the term `Locked(t, x)`, which itself is an object of type state. This is all made to work by the exisistence of the following implicit function, which lifts a `state` object to a Boolean, being true only if the object is in the *state soup*, represented by the variable *states*:

```
implicit def convState2Boolean(s: state): Boolean = states contains s
```

Two Kinds of State Producing Functions.

In the `Locked(t: Int, x: Int)` case class above, we saw a call of a `hot` function. Although it looks like the `hot` function we saw earlier, it is actually a different one, not returning a state, but updating the state it is called from within. That is, the `state` class provides the following functions, analog to the previous ones, but which just updates the state's transition function (return type is `Unit` and not `state`):

```
def always(ts: Transitions): state // always check the transitions
def watch(ts: Transitions): state  // watch until one of the transitions fi
def hot(ts: Transitions): state   // non-final, otherwise same meaning as w
def next(ts: Transitions): state   // non-final, one of the transitions mus
def wnext(ts: Transitions): state  // one of the transtions must fire next,
```

A user does not need to think about this distinction between the two sets of functions, once the specfication patterns have become familiar.

The Start-Stop State Machine using Rules

In this example we illustrate, using the start-stop example, how case classes can be used to represent a state machine with a loop. This is an alternative to the use of functions we saw earlier.

```
class TestMonitor extends Monitor[TaskEvent] {

  case class Start(task: Int) extends fact {
    wnext {
      case start(`task`) => Stop(task)
    }
  }

  case class Stop(task: Int) extends fact {
    next {
      case stop(`task`) => Start(task + 1)
    }
  }

  Start(0)
}
```

Both solutions in this case work equally well from a monitoring point of view. However, the use of case classes (facts) has one advantage: if we turn on printing mode, setting a monitor's `PRINT` flag to true (see below), then case class states will be printed nicely, whereas this is not the case when using anonymous states (using the function approach).

More Rule-Based Programming

This section illustrates some of the more esoteric functions, namely `exists` and `map`, for searching the *state soup* for states that satisfy certain conditions. These functions have similarities to Scala's functions with the same names.

The Property

Let's first sketch the property we want to monitor.

- *"At most one task can acquire a lock at a time. A task cannot release a lock it has not acquired."*

We provide two alternative formulations of this property, one using the `exists` function and one using the `map` function. These functions have the following signatures (the `map` function returns an object on which the `orelse` function is defined):

```
def exists(pred: PartialFunction[state, Boolean]): Boolean
def map(pf: PartialFunction[state, Set[state]]) orelse (set: => Set[state])
```

The `exists` function searches the *state soup* for a state that satisfies the partial function predicate `pred` provided as argument, and returns true of one that matches is found.

The `map` function searches the *state soup* for states mathcing the partial function `pf`, and returns the union of applying `pf` to all these states. Alternatively if no matching state is found it returns the `set` provided as second argument.

The Monitor using the exists Function

The idea is to add a state `Locked(t, x)` to the *state soup* when a lock is acquired. This state removes itself on a proper `release` event by the same thread of the same lock. Upon acquisition of a lock, we only add such a `Locked(t, x)` event if no other thread already holds the lock (another `Locked(_,x)` state exists in the *state soup*).

```
class MonitorUsingExists extends Monitor[LockEvent] {
  case class Locked(t: Isnt, x: Int) extends fact {
    watch {
      case release(`t`, `x`) => ok
    }
  }

  always {
    case acquire(t, x) =>
      if (exists {case Locked(_, `x`) => true}) error else Locked(t, x)
    case release(t, x) => ensure(Locked(t,x))
  }
}
```

The `ensure(b: Boolean): state` function returns the state `ok` if the Boolean condition `b` is true, otherwise it returns the state `error`.

The Monitor using the map Function

```
class MonitorUsingMap extends Monitor[LockEvent] {
  case class Locked(t: Int, x: Int) extends fact {
    watch {
      case release(`t`, `x`) => ok
    }
  }

  always {
    case acquire(t, x) => {
      map {
        case Locked(_,`x`) => error
      } orelse {
        Locked(t, x)
      }
    }
    case release(t, x) => ensure(Locked(t, x))
  }
}
```

Verifying Real-Time Properties

Daut supports monitoring of some forms of timing properties. We shall consider the lock acquisition and release scenario again, but slightly modified such that events carry a time stamp.

We shall then formulate a property about these time stamps.

The Events

The events now carry an additional time stamp `ts` indicating when the lock was acquired, respectively released:

```
trait LockEvent
case class acquire(t:Int, x:Int, ts:Int) extends LockEvent
case class release(t:Int, x:Int, ts:Int) extends LockEvent
```

The Property

The property now states that locks should be released in a timely manner:

- *"Property ReleaseWithin: A task acquiring a lock should eventually release it within 500 milliseconds."*

The Monitor

The monitor can be formulated as a monitor class parameterized with the number of time units within which a lock must be released after it has been acquired:

```
class ReleaseWithin(limit: Int) extends Monitor[LockEvent] {
  always {
    case acquire(t, x, ts1) =>
      hot {
        case release(`t`,`x`, ts2) => ensure(ts2 - ts1 <= limit)
      }
  }
}
```

The function `ensure(b: Boolean): state` returns either the `ok` state or the `error` state, depending on whether the Boolean condition `b` is true or not.

This can actually be expressed slightly more elegantly, leaving out the call of `ensure` and just provide its argument, as in:

```
class ReleaseWithin(limit: Int) extends Monitor[LockEvent] {
  always {
    case acquire(t, x, ts1) =>
      hot {
        case release(`t`,`x`, ts2) => ts2 - ts1 <= limit
      }
  }
}
```

This is due to the before mentioned implicit function:

```
implicit def convBoolean2StateSet(b: Boolean): Set[state] = Set(if (b) ok e
```

Finally, it should be mentioned that one can call the function `check(b: Boolean): Unit` at any point. It will report an error in case the Boolean condition `b` is false, but otherwise will let the monitor progress as if no error had occurred.

The Main Program

Finally, we can instantiate the monitor:-

```
object Main {
  def main(args: Array[String]) {
    val m = new ReleaseWithin(500) printSteps()
    m.verify(acquire(1, 10, 100))
    m.verify(release(1, 10, 800)) // violates property
    m.end()
  }
}
```

PS: note the call of `printSteps()` on the monitor (which returns the monitor). It has the same effect as adding `m.PRINT = true`.

Limitations of this Approach

The above described approach to the verification of timing properties is limited in the sense that it cannot detect a timing violation as soon as it occurs. For example, after an `acquire` event, if no `release` event occurs within the time window, the monitor will not know until either a proper `release` occurs, perhaps much later, or until the `end()` method is called and we find ourselves in the hot state. This can only be improved upon by e.g. introducing other more regularly occuring events containing time stamps and then include those in the specification.

The TraceContract system has timers internal to the monitor, but Daut does currently not support this.

Code in Specs and Invariants

This example illustrates the use of Scala code as part of a specification, and the use of class invariants, using the lock acquisition and release scenario again. First we reformulate the property we want to monitor.

The Property

- *"A task acquiring a lock should eventually release it. At most one task can acquire a lock at a time. At most 4 locks should be acquired at any moment"*.

The Monitor

The monitor declares a monitor local integer variable `count`, keeping track of the number of un-released locks. A monitor class invariant expresses the upper limit on the number of un-released locks.

```
class AcquireReleaseLimit extends Monitor[LockEvent] {
  var count : Int = 0

  invariant {count <= 4}

  always {
    case acquire(t, x) =>
      count +=1
      hot {
        case acquire(_,`x`) => error
        case release(`t`,`x`) => count -= 1; ok
      }
  }
}
```

The `invariant` function takes as argument a Boolean expression (call by name) and ensures that this expression is evaluated after each event processed by the monitor. If the expression ever becomes false, an error is issued.

The example illustrates how the temporal logic can be combined with programming. This paradigm of combining programming and temporal logic/state machines can of course be carried much furher.

Monitor Hierarchies

Monitors can be combined into a single monitor. For example, the following is possible:

```
class AcquireRelease extends Monitor[Event] {
  always {
    case acquire(t, x) =>
      hot {
        case acquire(_,`x`) => error
        case release(`t`,`x`) => ok
      }
  }
}

class ReleaseAcquired extends Monitor[Event] {
  case class Locked(t:Int, x:Int) extends fact {
    watch {
      case release(`t`,`x`) => ok
    }
  }

  always {
    case acquire(t,x) => Locked(t,x)
    case release(t,x) if !Locked(t,x) => error
  }
}

class Monitors extends Monitor[Event] {
  monitor(new AcquireRelease, new ReleaseAcquired) // monitor submonitors
}

object Main {
  def main(args: Array[String]) {
    val m = new Monitors // now monitoring two submonitors
    m.verify(acquire(1, 10))
    m.verify(release(1, 10))
    m.end()
  }
}
```

This allows to build a hierarchy of monitors, which might be useful for grouping.

Monitor Networks

Above we saw how monitors can be combined in a modular manner to form a hierarchy, with the sole purpose to just group monitors, without any added semantics. We here describe an alternative way of combining monitors in a network, where one monitor can send events to other monitors. We call such other monitors for *abstract monitors*, since the typical usecase is that a "concrete" monitor receives lower level events, and then sends higher level events to the abstract monitor when a certain sequence of lower level events has been detected. The following example illustrates this.

First we define the abstract monitor, including the abstract events we want to send to it. It checks that different commands always have different task ids or command numbers:

```
// Abstract events:

sealed trait AbstractEvent
case class Command(taskId: Int, cmdNum: Int) extends AbstractEvent

class AbstractMonitor extends Monitor[AbstractEvent] {
  always {
    case Command(taskId1, cmdNum1) => always {
      case Command(taskId2, cmdNum2) =>
        ensure (taskId1 != taskId2 || cmdNum1 != cmdNum2)
    }
  }
}
```

So what are commands? they are here generated by a lower level monitor, which monitors lower level events, which together represent a command execution. This concrete monitor creates an instance of the abstract monitor and sends `Command` events to it.

```
// Concrete events:

sealed trait ConcreteEvent
case class DispatchRequest(taskId: Int, cmdNum: Int) extends ConcreteEvent
case class DispatchReply(taskId: Int, cmdNum: Int) extends ConcreteEvent
case class CommandComplete(taskId: Int, cmdNum: Int) extends ConcreteEvent

class ConcreteMonitor extends Monitor[ConcreteEvent] {
  val abstractMonitor = monitorAbstraction(AbstractMonitor())

  always {
    case DispatchRequest(taskId, cmdNum) =>
      hot {
        case DispatchReply(`taskId`, `cmdNum`) =>
          hot {
            case CommandComplete(`taskId`, `cmdNum`) =>
              abstractMonitor(Command(taskId, cmdNum))
              println(s"command $taskId $cmdNum")
          }
      }
  }
}
```

The abstract monitor is created by a call of the `monitorAbstraction` method. This call ensures that when `end()` is called on the concrete monitor, it is also automatically called on the abstract monitor. It would also be possible to write:

```
val abstractMonitor = AbstractMonitor()
```

This would have the exact same effect, except that `end()` would not be automatically called.

Note: multiple monitors can send to the same abstract monitor.

Wrt. examples:

- See here for a complete example
- See here for an example of three monitors calling each other in a circular manner

Wrt. the last example, monitors calling each other in a recursive manner can be problematic. A better way might be to embed monitors in actors:

- See here

Using Indexing to Speed up Monitors

Indexing is an approach to speed up monitoring by defining a function from events to keys, and using the keys as entries in a hashmap to obtain only those states that are relevent to the particular event. This can be useful if our state soup ends up containing many (thousands) of states. The larger the number of states in the state soup, the more important indexing becomes for obtaining an efficient monitor. The improvement in speed can be several orders of magnitudes.

Let us illustrate the indexing approach with a slight modification of our locking example.

Events

We shall use the same events as before, except that we add an additional `CANCEL` event which releases all locks (it has the same meaning as a sequence of `release` calls on all acquisitions):

```
trait LockEvent
case class acquire(t: Int, x: Int) extends LockEvent
case class release(t: Int, x: Int) extends LockEvent
case object CANCEL extends LockEvent
```

The Property

Our property is the same as in our first example:

- *"A task acquiring a lock should eventually release it. At most one task can acquire a lock at a time"*.

The Traditional Monitor Approach

Let's do things a bit differenty and give names to the different monitor strategies. We first define our traditional generic monitor, calling it `SlowLockMonitor`, and the define our monitor as extending that (slightly modified to take `CANCEL` into account):

```
class SlowLockMonitor extends Monitor[LockEvent]

class CorrectLock extends SlowLockMonitor {
  always {
    case acquire(t, x) =>
      hot {
        case acquire(_, `x`) => error
        case CANCEL | release(`t`, `x`) => ok
      }
  }
}
```

Let us apply this monitor as follows:

```
object Main {
  def main(args: Array[String]) {
    val m = new CorrectLock
    m.verify(acquire(1, 100))
    m.verify(acquire(2, 200))
    m.verify(acquire(1, 200)) // error
    m.CANCEL
    m.end()
  }
}
```

The third event will violate the monitor since task 1 wants to acquire lock 200, which has already been acquired by task 2. Assume that we observed 100,000 lock acquisitions, then for each new lock acquisition we would at that point have to explore 100,000 states to see of there is a violation. We can optimize this and use indexing.

Indexing

Note that this property is **lock centric**: we can maintain a set of states for each lock: those states that concern only that lock. This is the idea in indexing. We can use the lock id as key in a mapping from keys to sets of states. This is done by overriding the `keyOf` function in the `Monitor` class (something the user has to do explicitly). This function has the type:

```
def keyOf(event: LockEvent): Option[Any]
```

This function takes an event as argument and returns an optional index of type `Any` (any index can be used). It's default return value is `None`. The function can be overridden as follows for our example (note why it may be a good idea to separate out the definition of `keyOf` from the actual monitor: we achive modularity and potential reuse of the `keyOf` function for other monitors):

```
class FastLockMonitor extends Monitor[LockEvent] {
  override def keyOf(event: LockEvent): Option[Int] = {
    event match {
      case acquire(_, x) => Some(x)
      case release(_, x) => Some(x)
      case CANCEL => None
    }
  }
}

class CorrectLock extends FastLockMonitor {
  // as before:
  always {
    case acquire(t, x) =>
      hot {
        case acquire(_, `x`) => error
        case CANCEL | release(`t`, `x`) => ok
      }
  }
}
```

The `keyOf` function here extracts the lock `x` from the event and turns it into the key `Some(x)`. This now means that all events concerning a specific lock `x` are fed to only those states concerned with that lock.

Note that `CANCEL` is mapped to `None`. This has as effect that this event is sent to all states, independent of their key. This is exaxtly what we want: all locks should be released.

Be Careful with the Definition of `keyOf`

One should be careful with the definition of the `keyOf` function (similarly to how one has to be careful when defining the functions `hashCode` and `equals`). Suppose for example that we instead had defined the `keyOf` function as follows, where the key is the **task id**, and not the lock id:

```
class BadLockMonitor extends Monitor[LockEvent] {
  override def keyOf(event: LockEvent): Option[Int] = {
    event match {
      case acquire(t, _) => Some(t)
      case release(t, _) => Some(t)
      case CANCEL => None
    }
  }
}

class CorrectLock extends BadLockMonitor {
  // as before:
  always {
    case acquire(t, x) =>
      hot {
        case acquire(_, `x`) => error
        case CANCEL | release(`t`, `x`) => ok
      }
  }
}
```

In this case, our monitor will not detect the violation in the sequence:

```
    m.verify(acquire(2, 200))
    m.verify(acquire(1, 200)) // error
```

since the first event is sent to the state set denoted by 2 and the second event is sent to the state set denoted by 1. Those sets are different, and hence no violation is detected.

Using Indexing to Check Past Time Properties

An interesting application of indexing, beyond just optimization, is the formulation of past time properties, e.g.: *"if some event P happens now then some other event Q should have happened in the past"*. Recall that we previously used a fact (objects of case classes, specifically the fact `Locked(t,x)`), to model the fact that thread `t` has acquired lock `x`. We can instead use the indexing feature that all events concerning e.g. a lock are sent to the same state set, and we can therefore require that a `release` is not allowed for a lock before it has been acquired. Let's see how this looks like.

The property we formulated earlier was the following:

- *"A task acquiring a lock should eventually release it. At most one task can acquire a lock at*

> *a time. A task cannot release a lock it has not acquired."*.

It is the last requirement *"A task cannot release a lock it has not acquired"*, that is a past time property: if a task is released, it must have been acquired in the past, and not released since. We shall leave out the `CANCEL` event from this example. The monitor for this property, using indexing, can be formulated as follows.

```
class AcquireRelease extends Monitor[LockEvent] {
  override def keyOf(event: LockEvent): Option[Int] = {
    event match {
      case acquire(_, x) => Some(x)
      case release(_, x) => Some(x)
    }
  }

  def start(): state =
    watch {
      case acquire(t, x) => hot {
        case acquire(_, `x`) => error
        case release(`t`, `x`) => start()
      }
      case release(_, _) => error
    }

  start()
}
```

The `keyOf` function extracts the lock id for each event, hence all events for one particular lock are sent to the same state set. The monitor itself is a state machine with one named state `start`, and an anonymous hot state, from which there is a transition back to the `start` state when a lock is released. Note the transition:

```
case release(_, _) => error
```

as part of the `start` state, which will trigger an error in case a `release` event arrives before an `acquire` event for a lock.

This is fundamentally how slicing-based systems such as MOP model properties.

Writing Textbook Automata using Indexing

In the above example, we used the states `hot` and `watch`. These are states with a UMLlike semantics: we stay in these states until we see an event that matches one of the outgoing transitions. If we instead use the states `wnext` (if there is a next event, it has to match one of the transitions) and `next` (there has to be a next event it it has to match one of the transitions), we can formulate a monitor without having to indicate the wrong transitions explicitly, it follows indirectly from the semantics of `wnext` and `next`, just like in classical automata theory:

```
class AcquireReleaseTextBookLogic extends Monitor[LockEvent] {
  override def keyOf(event: LockEvent): Option[Int] = ... // as above

  def doAcquire(): state =
    wnext {
      case acquire(t, x) => next {
        case release(`t`, `x`) => doAcquire()
      }
    }

  doAcquire()
}
```

We can also name the inner anonymous state, let us name it `doRelease`, arriving the following automaton, with equivalent semantics:

```
class AcquireReleaseTextBookAutomaton extends Monitor[LockEvent] {
  override def keyOf(event: LockEvent): Option[Int] = ... // as above

  def doAcquire(): state =
    wnext {
      case acquire(t, x) => doRelease(t, x)
    }

  def doRelease(t: Int, x: Int) =
    next {
      case release(`t`, `x`) => doAcquire()
    } label(t, x)

  doAcquire()
}
```

Note: One has to ensure that only events relevant for the property are submitted to the monitor in order to use the states `wnext` and `next`.

The stay Directive

When using indexing combined with `wnext` and `next` states, it can be necessary to explicitly indicate that a certain event in a certain state is acceptable and just causes the monitor to stay in that state, corresponding to a self-loop in automaton theory (a transition where the source state and the target state is the same). Let us modify the requirement to allow reentrant locks, meaning that a thread that a task that currently holds the lock can acquire it again without causing a deadlock.

- *"A task acquiring a lock should eventually release it. At most one task can acquire a lock at a time. Howevever, locks are reentrant. A task cannot release a lock it has not acquired."*.

This can be expressed using `stay` as "target state", which means the current state we are in.

```
class AcquireReleaseTextBookStayAutomaton extends Monitor[LockEvent] {
  override def keyOf(event: LockEvent): Option[Int] = ... // as above

  def doAcquire(): state =
    wnext {
      case acquire(t, x) => doRelease(t, x)
    }

  def doRelease(t: Int, x: Int) =
    next {
      case acquire(`t`, `x`) => stay // the lock is reentrant
      case release(`t`, `x`) => doAcquire()
    } label(t, x)

  doAcquire()
}
```

Note: The `stay` state can also be used in other context (without indexing). It can e.g. be used if some side effect is needed, but we want to stay in the current state.

Monitoring Only Relevant Events

The approach described just above only works if the events submitted to the monitor are exactly the events mentioned in the automaton. If this is not the case we have to filter out the irrelevant events. Consider for example that the `LockEvent` type was defined as originally, containing also a `CANCEL` event:

```
trait LockEvent
case class acquire(t: Int, x: Int) extends LockEvent
case class release(t: Int, x: Int) extends LockEvent
case object CANCEL extends LockEvent
```

However, we do not want to take this event into consideration. For the above automaton to work we must filter out the `CANCEL` event. This can be done by overriding the `relevant` method, as done in the following version of the monitor.

```
class AcquireReleaseTextBookLogic extends Monitor[LockEvent] {
  override def keyOf(event: LockEvent): Option[Int] = ... // as above

  override def relevant(event: LockEvent): Boolean = {
    event match {
      case acquire(_, _) | release(_, _) => true
      case _ => false
    }
  }

  def doAcquire(): state =
    wnext {
      case acquire(t, x) => doRelease(t, x)
    }

  def doRelease(t: Int, x: Int) =
    next {
      case release(`t`, `x`) => doAcquire()
    }

  doAcquire()
}
```

How to React to Errors

Error handing in monitors can be done in a number of ways.

Checing on Final Error Count

Once monitoring is done (assuming `end()` is called, but even before), one can call the `getErrorCount` method on the monitor to determine how many errors occurred, and if any, perform a proper action:

```
object Main {
  def main(args: Array[String]) {
    val m = new Monitors // now monitoring two submonitors
    m.verify(acquire(1, 10))
    m.verify(release(1, 10))
    m.end()
    if (m.getErrorCount > 0) { /* action */ }
  }
}
```

Writing Error Handling Code in Transitions

One can of course write error handling code in the transitions themselves, as in:

```
class AcquireRelease extends Monitor[Event] {
  def handleError() { /* action */ }

  always {
    case acquire(t, x) =>
      hot {
        case acquire(_,`x`) =>
          handleError() // handle the error
          error // and return the error state
        case release(`t`,`x`) => ok
      }
  }
}
```

Using Callback Function

Finally, one can override the `callBack()` method defined in the `Monitor` class. This method will be automatically called upon detection of an error. It does not need to be called explicitly.

```
class AcquireRelease extends Monitor[Event] {
  override def callBack() { /* action */ }

  always {
    case acquire(t, x) =>
      hot {
        case acquire(_,`x`) =>
          error // call callBack() and return the error state
        case release(`t`,`x`) => ok
      }
  }
}
```

Implementing New Temporal Operators

Daut is extensible in the sense that one can define new temporal operators/patterns. We shall here mention two ways of doing this, namely repsectively as a (monitor) class or as a function. We shall illustrate the concepts on an example concerning a radio, which can be opened and closed, and over which messages can be sent, and received at the other end.

The Events

The event type covers opening and closing the radio, as well as sending messages over the radio, which are then hopefully received at the other end:

```
trait RadioEvent
case object Open extends RadioEvent
case object Close extends RadioEvent
case class Send(msg: String) extends RadioEvent
case class Receive(msg: String) extends RadioEvent
```

The Property

The properties we want to check is the following two:

- *"An opened radio must eventually be closed."*
- *"When the radio is open, all sent messages should eventually be received."*

Each of these properties follows a pattern. The first is the *response* pattern: when an event occurs, some other event must occur later. The second is a *scope* pattern: in between two events (here `Open` and `Close`) some condition must hold.

A Pattern Can be Programmed as a Monitor

The first *response* pattern can be programmed as a parameterized monitor class, which checks a particular property dependent on the parameters provided to the class. This is done in the following:

```
class Response[E](e1: E, e2: E) extends Monitor[E] {
  always {
    case `e1` => hot {
      case `e2` => ok
    }
  }
}
```

It can then be applied as follows: `new Response(Open, Close)` to create a monitor checking that an `Open` is always followed by a `Close` (before `end()` is called). We shall see this below.

A Pattern can be Programmed as a Function

The second *scope* pattern is programmed as a function that takes two events and a transition function (partial function from events to sets of traces) as arguments:

```
class NewMonitor[E] extends Monitor[E] {
  def between(e1: E, e2: E)(tr: Transitions): state = {
    always {
      case `e1` =>
        unless {
          case `e2` => ok
        } watch (tr)
    }
  }
}
```

The idea is that any monitor that uses this function will now have to **extend** `NewMonitor` instead of `Monitor`, and then call this function, a we shall illustrate below.

The function returns a state, which behaves as indicated: it is always the case, that if `e1` is observed then the transition `tr` will be applied (if matching) to every incoming event, unless and until `e2` occurs (not required to occur). The `unless` function has the signature:

```
def unless(ts1: Transitions) watch (ts2: Transitions): state
```

and checks `ts2` repeatedly unless `ts1` applies (which does not need to happen). There is a similar:

```
def until(ts1: Transitions) watch (ts2: Transitions): state
```

which requires `ts1` to eventually trigger.

Note that whatever monitoring is initiated in `tr` betweeen `e1` and `e2` will continue if needed after `e2`. Writing a function that stops `tr` after `e2` is a non-trivial programming exercise left to the reader.

The Monitors

We can now create a monitor for the second property by extending the `NewMonitor` class and call the function `between`, which creates a state, and as we have explained earlier, the first created state becomes an initial state (as a side effect).

```
class ReceiveWhenOpen extends NewMonitor[RadioEvent] {
  between(Open,Close) {
    case Send(m) => hot {
      case Receive(`m`) => true
    }
  }
}
```

We now create a monitor containing this monitor and an instance of the `Response` monitor as submonitors:

```
class AllMonitors extends Monitor[RadioEvent] {
  monitor(
    new Response(Open,Close),
    new ReceiveWhenOpen
  )
}
```

The Main Program

Finally, we can invoke `AllMonitors`:

```
object Main {
  def main(args: Array[String]) {
    val m = new AllMonitors
    m.verify(Send("ignore this message"))
    m.verify(Open)
    m.verify(Send("hello"))
    m.verify(Send("world"))
    m.verify(Send("I just saw a UFO!")) // violating since not received
    m.verify(Receive("hello"))
    m.verify(Close)
    m.verify(Receive("world"))
    m.verify(Send("and ignore this one too"))
    m.end()
  }
}
```

Options

Daut offers three option variables that can be set:

- `DautOptions.DEBUG` (static variable): when set to true, causes each monitor step to be printed, including event and resulting set of states. Default is false.

- `DautOptions.DEBUG_ALL_EVENTS` (static variable): when set to true and `DEBUG` is true, causes all events to be reported. If false, only events triggering transitions are shown. Default is false.

- `DautOptions.DEBUG_TRACES` (static variable): when set to true and `DEBUG` is true, causes the trace that lead to a state be printed as part of the state. Default is true.

- `DautOptions.PRINT_ERROR_BANNER` (static variable): when set to true, when an error occurs, a very big ERROR BANNNER is printed (to make it visible amongst plenty of output). Default is false.

- `DautOptions.RECORD_OK` (static variable): when set to true, every `ok` (success) reached will be reported. Default is false.

- `DautOptions.SHOW_TRANSITIONS` (static variable): when set to true, events that trigger transitions are shown. Default is false.

- `Monitor.STOP_ON_ERROR` : when set to true an error will case the monitor to stop. Default is false. This option is local to each monitor.

These options can be set as shown in the following example:

```
DautOptions.DEBUG = true
DautOptions.PRINT_ERROR_BANNER = false
val m = MyMonitor()
m.STOP_ON_ERROR = true
...
```

Labeling of Anonymous States for Debugging Purposes

When debugging a monitor with the `DautOptions.DEBUG` flag set to true, states will be printed on standard out. For anonymous states we will only get printed what kind of state it is (`always` , `hot` , ...). We can added information to be printed with the `label` function, for example as follows:

```
  always {
    case acquire(t, x) =>
      hot {
        case acquire(_, `x`) => error
        case CANCEL | release(`t`, `x`) => ok
      } label(t, x)
  }
```

This will cause `hot(1,2)` to be printed instead of just `hot` (for values `t` =1 and `x` =2).

It is also, as an alternative, possible to provide the label information to the anonymous state producing functions (`always` , `hot` , ...) as follows, with the exact same effect on the debugging information produced as the example above:

```
  always {
    case acquire(t, x) =>
      hot(t,x) {
        case acquire(_, `x`) => error
        case CANCEL | release(`t`, `x`) => ok
      }
  }
```

Example of using debugging mode

The slow monitor not using indexing

Consider the example introduced in the section `Using Indexing to Speed up Monitors` above. We shall run this example in debug mode and explain the resulting output. We first debug the slow monitor:

```
class CorrectLock extends SlowLockMonitor {
  always {
    case acquire(t, x) =>
      hot {
        case acquire(_, `x`) => error
        case CANCEL | release(`t`, `x`) => ok
      } label(t, x)
  }
}
```

Note that we have added the `label(t,x)` to the `hot` state. This will cause the debug output to contain terms such as `hot(1,100)`. Next, in the main program, we set the `DEBUG` flag to true:

```
object Main {
  def main(args: Array[String]) {
    val m = new CorrectLock
    DautOptions.DEBUG = true
    m.verify(acquire(1, 100))
    m.verify(acquire(2, 200))
    m.verify(acquire(1, 200)) // error
    m.verify(CANCEL)
    m.end()
  }
}
```

When we run this program we get the following output:

```
===[acquire(1,100)]===

--- CorrectLock:
[memory]
  hot(1,100)
  always

===[acquire(2,200)]===

--- CorrectLock:
[memory]
  hot(1,100)
  hot(2,200)
  always

===[acquire(1,200)]===

███████╗██████╗ ██████╗  ██████╗ ██████╗
██╔════╝██╔══██╗██╔══██╗██╔═══██╗██╔══██╗
█████╗  ██████╔╝██████╔╝██║   ██║██████╔╝
██╔══╝  ██╔══██╗██╔══██╗██║   ██║██╔══██╗
███████╗██║  ██║██║  ██║╚██████╔╝██║  ██║
╚══════╝╚═╝  ╚═╝╚═╝  ╚═╝ ╚═════╝ ╚═╝  ╚═╝

CorrectLock error # 1

--- CorrectLock:
[memory]
  hot(1,100)
  hot(1,200)
  always

===[CANCEL]===

--- CorrectLock:
[memory]
  always


Ending Daut trace evaluation for CorrectLock
```

For each event we get a line indicating what event, e.g `===[acquire(1,100)]===`. Next, for each monitor, here we have one: `CorrectLock`, we see what states are maintained in the `main` state set. For example after the events `acquire(1,100)` and `acquire(2,200)` we have the set:

```
[memory]
  hot(1,100)
  hot(2,200)
  always
```

consisting of two `hot` states, and the initial `always` state. As we can see, all the states are in the same set. After the `CANCEL` event we see that this set returns to only contain the `always` state.

The fast monitor using indexing

If we now instead run the example with the fast indexed monitor:

```
class CorrectLock extends FastLockMonitor {
  ...
}
```

we get the following output:

```
===[acquire(1,100)]===

--- CorrectLock:
[memory]
  always

[index=100]
  hot(1,100)
  always

===[acquire(2,200)]===

--- CorrectLock:
[memory]
  always

[index=100]
  hot(1,100)
  always
[index=200]
  hot(2,200)
  always
```

```
===[acquire(1,200)]===

███████╗██████╗ ██████╗  ██████╗ ██████╗
██╔════╝██╔══██╗██╔══██╗██╔═══██╗██╔══██╗
█████╗  ██████╔╝██████╔╝██║   ██║██████╔╝
██╔══╝  ██╔══██╗██╔══██╗██║   ██║██╔══██╗
███████╗██║  ██║██║  ██║╚██████╔╝██║  ██║
╚══════╝╚═╝  ╚═╝╚═╝  ╚═╝ ╚═════╝ ╚═╝  ╚═╝


CorrectLock error # 1

--- CorrectLock:
[memory]
  always

[index=100]
  hot(1,100)
  always
[index=200]
  hot(1,200)
  always

===[CANCEL]===

--- CorrectLock:
[memory]
  always

[index=100]
  always
[index=200]
  always

Ending Daut trace evaluation for CorrectLock
```

Note how the different `hot` states are now distributed in different buckets in the indexed hash map. Note also how the `always` state is copied to these buckets and is always active.

Not using labels

Suppose we do not label the anonymous `hot` state, as in the following where the label has been commented out:

```
class CorrectLock extends FastLockMonitor {
  always {
    case acquire(t, x) =>
      hot {
        case acquire(_, `x`) => error
        case CANCEL | release(`t`, `x`) => ok
      } // label(t, x)
  }
}
```

Then the output will be less informative, and will only tell us what kind of states (`always` , `hot` , ...) are in the state soups, without any values, as follows:

```
===[acquire(1,100)]===

--- CorrectLock:
[memory]
  always

[index=100]
  hot
  always

===[acquire(2,200)]===

--- CorrectLock:
[memory]
  always

[index=100]
  hot
  always
[index=200]
  hot
  always

===[acquire(1,200)]===

███████╗██████╗ ██████╗  ██████╗ ██████╗
██╔════╝██╔══██╗██╔══██╗██╔═══██╗██╔══██╗
█████╗  ██████╔╝██████╔╝██║   ██║██████╔╝
██╔══╝  ██╔══██╗██╔══██╗██║   ██║██╔══██╗
███████╗██║  ██║██║  ██║╚██████╔╝██║  ██║
╚══════╝╚═╝  ╚═╝╚═╝  ╚═╝ ╚═════╝ ╚═╝  ╚═╝
```

```
CorrectLock error # 1

--- CorrectLock:
[memory]
  always

[index=100]
  hot
  always
[index=200]
  hot
  always

===[CANCEL]===

--- CorrectLock:
[memory]
  always

[index=100]
  always
[index=200]
  always

Ending Daut trace evaluation for CorrectLock
```

Other Helper Functions

The `Monitor` class provides a collection of methods which can help viewing the results of a run. These are explained in the following

Recording Messages

The following method allows to add arbitrary text messages as recordings in a monitor:

```
def record(message: String): Unit
```

At any point in time, the current recordings (including error messages) of a monitor, and all its submonitors, can be extracted with the method:

```
def getRecordings(): List[String]
```

Printing Monitor States

The following method prints the internal memory of a monitor:

```
def printStates(): Unit
```

Printing Selected Triggering Events on Standard Out

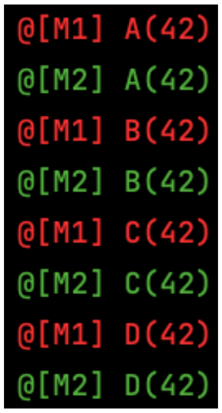
There are a number of options for showing and recording which events cause transitions to trigger in monitors.

The following method when called on a monitor object with `flag` being true (default parameter value) will cause events to be printed when triggering transitions in the monitor, and all its submonitors.

```
def showTransitions(flag: Boolean = true): Monitor[E]
```

Note, however, that the submonitors must have been added already for a call of this method to have effect on the submonitors.

Events will be printed out in color with different colors chosen for different monitors (in a circular manner since the number of colors is limited). The output will look something like the following, here for two monitors `M1` and `M2`, with the monitor name in square brackets and the event following.



```
@[M1] A(42)
@[M2] A(42)
@[M1] B(42)
@[M2] B(42)
@[M1] C(42)
@[M2] C(42)
@[M1] D(42)
@[M2] D(42)
```

It is possible to control how events are reported by overriding the following method:

```
protected def renderEventAs(event: E): Option[String] = None
```

By default, when applied to an event `e`, it returns `None`, which means that the event will be printed as the default `e.toString()`. The user can override the method to instead return

`Some(s)` for various events, resulting in the event as being rendered as `s` instead. This can e.g. be used to highlight certain arguments to the event, or/and filter out arguments.

Printing All Triggering Events on Standard Out

An alternative is to set the following variable to true, which will cause all events that trigger a transition, in any monitor, to be printed (`Monitor` is an object).

```
DautOptions.SHOW_TRANSITIONS: Boolean = true
```

Writing Events as JSON Objects to Permanent Memory

Finally, it is possible to cause selected events that trigger transitions to be written in JSON format to a file.

Assume we have defined the following events and monitor.

```
sealed trait ConcreteEvent
case class DispatchRequest(taskId: Int, cmdNum: Int) extends ConcreteEvent
case class DispatchReply(taskId: Int, cmdNum: Int) extends ConcreteEvent
case class CommandComplete(taskId: Int, cmdNum: Int) extends ConcreteEvent

class ConcreteMonitor extends Monitor[ConcreteEvent] {
  always {
    case DispatchRequest(taskId, cmdNum) =>
      hot {
        case DispatchRequest(`taskId`, `cmdNum`) => error
        case DispatchReply(`taskId`, `cmdNum`) =>
          hot {
            case CommandComplete(`taskId`, `cmdNum`) => ok
          }
      }
  }
}
```

First we have to define an encoder function, which maps events to string representations of JSON objects. One way is to use the json4s library. This function will below be provided as argument to a Daut function. Note that if this function returns `None` for an event, this event will not be written to permanent memory as a JSON object.

```
import org.json4s._
import org.json4s.native.Serialization
import org.json4s.native.Serialization.write


implicit val formats: Formats = Serialization.formats(NoTypeHints)

def encoder(obj: Any): Option[String] = {
  val map = obj match {
    case DispatchRequest(taskId, cmdNum) =>
      Map("kind" -> "DispatchRequest", "name" -> taskId, "cmdNum" -> cmdNum
    case DispatchReply(taskId, cmdNum) =>
      Map("kind" -> "DispatchReply", "name" -> taskId, "cmdNum" -> cmdNum)
    case CommandComplete(taskId, cmdNum) =>
      Map("kind" -> "CommandComplete", "name" -> taskId, "cmdNum" -> cmdNum
    case _ => return None
  }
  Some(write(map))
}
```

We can now ask Daut to log events in a file as follows:

```
Monitor.logTransitionsAsJson("output.jsonl", encoder)
```

After that if we apply the monitor to a trace as follows:

```
val trace: List[ConcreteEvent] = List(
  DispatchRequest(1, 1),
  DispatchReply(1, 1),
  CommandComplete(1, 1)
)

val monitor = new ConcreteMonitor
monitor(trace)
```

a file `output.jsonl` will be created containing the following lines:

```
{"kind":"DispatchRequest","name":1,"cmdNum":1}
{"kind":"DispatchReply","name":1,"cmdNum":1}
{"kind":"CommandComplete","name":1,"cmdNum":1}
```

See daut48*log*json.Main for an example.

Using Piper Mode for JSONL Files

Daut can be applied to read JSON objects from an input file written to by a concurrentlty executing task (online monitoring).

See daut42_json.Main

To generate a script to run this program:

```
generateRunScript daut42_json.Main run42.sh
```

Now, one can execute:

```
cat src/test/scala/daut42_json/file1.jsonl| ./run42.sh
Read: {"id" :  "dispatch", "task_id" :  1, "cmd_nr" : 1, "cmd_type": "START
Read: {"id" :  "reply",   "task_id" :  1, "cmd_nr" : 1, "cmd_type": "START"
Read: {"id" :  "complete", "task_id" :  1, "cmd_nr" : 1, "cmd_type": "START
Read: {"id" :  "dispatch", "task_id" :  2, "cmd_nr" : 1, "cmd_type": "START
Read: {"id" :  "reply",   "task_id" :  2, "cmd_nr" : 1, "cmd_type": "START"
Read: {"id" :  "complete", "task_id" :  2, "cmd_nr" : 1, "cmd_type": "START
Read: {"id" :  "complete", "task_id" :  2, "cmd_nr" : 1, "cmd_type": "START

*** ERROR
trigger event: Dispatch(2,1,START) event number 4
current event: Complete(2,1,START) event number 7
CommandMonitor error # 1
```

Contributions

Daut was developed by Klaus Havelund (klaus.havelund@jpl.nasa.gov) with contributions by Nicolas Rouquette (nfr@jpl.nasa.gov).