A Scala API for Runtime Verification

Klaus Havelund
Jet Propulsion Laboratory
Pasadena, California
A Scala API for Runtime Verification

DSL

Klaus Havelund
Jet Propulsion Laboratory
Pasadena, California
understanding complex systems by analyzing their execution

A Scala API for Runtime Verification

DSL

Klaus Havelund
Jet Propulsion Laboratory
Pasadena, California
log analysis
fault protection

event

response

monitor
a DSL for log analysis

COMMAND("STOP_CAMERA", 1, 22:50.00)
COMMAND("ORIENT_ANTENNA_TOWARDS_GROUND", 2, 22:50.10)
SUCCESS("ORIENT_ANTENNA_TOWARDS_GROUND", 3, 22:52.02)
COMMAND("STOP_CAMERA", 4, 22:55.01)
SUCCESS("ORIENT_ANTENNA_TOWARDS_GROUND", 5, 22:56.19)
COMMAND("STOP_ALL", 6, 23:01.10)
FAIL("ORIENT_ANTENNA_TOWARDS_GROUND", 7, 23:02.02)
a LogScope property

CommandMustSucceed:

“An issued command must succeed, without a failure to occur before then”.

```dialogue
monitor CommandMustSucceed {
always {
    COMMAND(n,x) => RequireSuccess(n,x)
}

hot RequireSuccess(name,number) {
    FAIL(name,number) => error
    SUCCESS(name,number) => ok
}
}
```
rule_schema ::= 
   modifier+ "{" transition+ ""}"
   | modifier* ident ["(" ident,* ")"] ["{" transition+ ""}"]

modifier ::= 
   "init" | "always" | "step" | "next" | "hot"

transition ::= pattern,* "=>" pattern,*

pattern ::= ["!"] ident ["(" constraint,* ")"]

constraint ::= 
   ident "::" range
   | range
user reaction

excellent

• I read the manual and was up an running, all before lunch

• my first spec had no errors and just worked

but (2 days later)

• can I define a function and call it in a formula?

• is it possible to re-use formulas?
external versus internal DSL

- **LogScope**
  - DSL
  - parser
  - programming language

- **TraceContract**
  - DSL
  - programming language
  - combines parameterized state machines and temporal logic.

external DSL

internal DSL
pros and cons for internal DSL

**pros**

- decreases development effort
- increases expressiveness
- allows use of existing IDE, debuggers, etc.

**cons**

- steep learning curve
- limited analyzability
Introducing Scala

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages, enabling Java and other programmers to be more productive. Code sizes are typically reduced by a factor of two to three when compared to an equivalent Java application. Read more.

Scala 2.9.0 RC2

Created by admin on 2011-04-26, Updated: 2011-04-26, 15:35

The second release candidate of the new Scala 2.9 distribution is now available: Scala 2.9.0 RC2 is currently available from our Download Page. The Scala 2.9.0 codebase includes several additions, notably the new Parallel Collections, but it also introduces improvements on many existing features, and contains many bug fixes. Please help us with the testing of this release candidate, and let us know of any issues you may detect.

Login or register to post comments Read more

The Scala IDE for Eclipse beta 2 available now!

Scala as a unifier

- script-like
- object oriented
- functional
- high performance with strong typing
abstract class Event

case class COMMAND(name: String, nr: Int) extends Event

case class SUCCESS(name: String, nr: Int) extends Event

case class FAIL(name: String, nr: Int) extends Event

val trace : List[Event] =
List(
    COMMAND("STOP_DRIVING", 1),
    COMMAND("TAKE_PICTURE", 2),
    SUCCESS("TAKE_PICTURE", 2),
    SUCCESS("TAKE_PICTURE", 2)
)
monitor CommandMustSucceed {
  always {
    COMMAND(n,x) => RequireSuccess(n,x)
  }
}

hot RequireSuccess(name,number) {
  FAIL(name,number) => error
  SUCCESS(name,number) => ok
}
}

class CommandMustSucceed extends Monitor[Event] {
  always {
    case COMMAND(n,x) => RequireSuccess(n,x)
  }

  def RequireSuccess(name: String, number: Int) =
  hot {
    hot {
      case FAIL(`name`, `number`) => error
      case SUCCESS(`name`, `number`) => ok
    }
  }
}
monitor CommandMustSucceed { 
    always {
        COMMAND(n,x) => RequireSuccess(n,x)
    }

    hot RequireSuccess(name,number) {
        FAIL(name,number) => error
        SUCCESS(name,number) => ok
    }
}
monitor CommandMustSucceed {
  always {
    COMMAND(n, x) => RequireSuccess(n, x)
  }

  hot RequireSuccess(name, number) {
    FAIL(name, number) => error
    SUCCESS(name, number) => ok
  }
}
monitor CommandMustSucceed {
  always {
    COMMAND(n,x) => RequireSuccess(n,x)
  }

  hot RequireSuccess(name,number) {
    FAIL(name,number) => error
    SUCCESS(name,number) => ok
  }
}

first 10 commands must succeed

class CommandMustSucceed extends Monitor[Event] {
  var count = 0
  always {
    always {
      case COMMAND(n, x) if count < 10 =>
        count += 1
        not(FAIL(n, x)) until (SUCCESS(n, x))
    }
  }
}
class Monitor[Event] {

...  

  type Block = PartialFunction[Event, Formula] (*
\label{type-block}*)

  // states:
  def always(block: Block): Formula
  def state(block: Block): Formula
  def hot(block: Block): Formula
  def step(block: Block): Formula
  def strong(block: Block): Formula
  def weak(block: Block): Formula

  // future time temporal logic:
  def not(formula: Formula): Formula
  def globally(formula: Formula): Formula
  def eventually(formula: Formula): Formula
  def strongnext(formula: Formula): Formula
  def matches(predicate: PartialFunction[Event, Boolean]): Formula
  def within(time: Int)(formula: Formula): Formula
}

the **state** function

**CommandMustSucceed:**

“An issued command can succeed at most once”.

class MaxOneSuccess extends Monitor[Event] {
  always {
    case SUCCESS(_, number) =>
      state {
        case SUCCESS(_, `number`) => error
      }
  }
}

"An issued command can succeed at most once".
analyzing a trace

class Requirements extends Monitor[Event] {
    monitor(
        new CommandMustSucceed,
        new MaxOneSuccess
    )
}

object Apply {
    def readLog(): List[Event] = {...}

    def main(args: Array[String]) {
        val monitor = new Requirements
        val log = readLog()
        monitor.verify(log)
    }
}

compose

run
Monitor: CommandMustSucceed

Error trace:
  1=COMMAND(STOP_DRIVING,1)

Monitor: MaxOneSuccess

Error trace:
  2=COMMAND(TAKE_PICTURE,2)
  3=SUCCESS(TAKE_PICTURE,2)
  4=SUCCESS(TAKE_PICTURE,2)
class `Monitor[Event]` extends `DataBase` with `Formulas[Event]`

This class offers all the features of TraceContract. The user is expected to extend this class. The class is parameterized with the event type. See the the explanation for the `Tracecontract` package for a full explanation.

The following example illustrates the definition of a monitor with two properties: a safety property and a liveness property.

```scala
class Requirements extends Monitor[Event] {
  requirement('CommandMustSucceed) {
    case COMMAND(x) =>
      hot {
        case SUCCESS(x) => ok
      }
  }
  requirement('CommandAtMostOnce) {
    case COMMAND(x) =>
      state {
        case COMMAND('x) => error
      }
  }
}
```

Event: the type of events being monitored.

**Instance constructors**

- `new Monitor()`

**Type Members**

- `type Block = PartialFunction[Event, Formula]`
  - Defines the type of transitions out of a state.
  - `class BooleanOps extends AnyRef`
  - Generated by implicit conversion from `Boolean`.
  - `class ElsePart extends AnyRef`
  - The else part of an `if (condition) Then formula1 Else formula2`.
  - `class EventFormulaOps extends AnyRef`
  - Target if implicit conversion of events.
  - `class Fact extends AnyRef`
  - Facts to be added to and removed from the fact database.
  - `class FactOps extends AnyRef`
  - Operations on Facts.
  - `class Formula extends AnyRef`
  - Each different kind of formula supported by TraceContract is represented by an object or class that extends this class.
  - `class IntOps extends AnyRef`
  - Generated by implicit conversion from `integer`.
  - `class IntPairOps extends AnyRef`
  - Generated by implicit conversion from `integer pair`.
  - `class ThenPart extends AnyRef`
  - The then part of an `if (condition) Then formula1 Else formula2`.
  - `type Trace = List[Event]`
def error(message: string): formula
    Emits the error message provided as argument and evaluates to False.

def error: Formula
    Emits an error message and evaluates to False.

def eventually(formula: Formula): Formula
    Eventually true (an LTL formula).

def eventuallyBw(m: Int, n: Int, x: Int = 1)(formula: Formula): Formula
    Eventually true between m and n steps.

def eventuallyEq(n: Int)(formula: Formula): Formula
    Eventually true at step n.

def eventuallyGe(n: Int)(formula: Formula): Formula
    Eventually true at or after minimally n steps.

def eventuallyGt(n: Int)(formula: Formula): Formula
    Eventually true after n steps.

def eventuallyLe(n: Int)(formula: Formula): Formula
    Eventually true in maximally n steps.

def eventuallyLt(n: Int)(formula: Formula): Formula
    Eventually true in less than n steps.

def factExists(pred: PartialFunction[Fact, Boolean]): Boolean
    Tests whether a fact exists in the fact database, which satisfies a predicate.

def getMonitorResult: MonitorResult[Event]
    Returns the result of a trace analysis for this monitor.

def getMonitors: List[Monitor[Event]]
    Returns the sub-monitors of a monitor.

def globally(formula: Formula): Formula
    Globally true (an LTL formula).

def hot(m: Int, n: Int)(block: PartialFunction[Event, Formula]): Formula
    A hot state waiting for an event to eventually match a transition (required) between m and n steps.

def hot(block: PartialFunction[Event, Formula]): Formula
    A hot state waiting for an event to eventually match a transition (required).

def informal(name: Symbol)(explanation: String): Unit
    Used to enter explanations of properties in informal language.

def informal(explanation: String): Unit
    Used to enter explanations of properties in informal language.

def matches(predicate: PartialFunction[Event, Boolean]): Formula
    Matches current event against a predicate.

def monitor(monitors: Monitor[Event]*): Unit
    Adds monitors as sub-monitors to the current monitor.

def never(formula: Formula): Formula
    Never true (an LTL-inspired formula).

def not(formula: Formula): Formula
    Boolean negation.

def ok(message: String): Formula
    Emits the message provided as argument and evaluates to True.

def ok: Formula
    Equivalent to True.
def eventuallySt(n: Int)(formula: Formula): Formula
    Eventually true after n steps.

def eventuallyLe(n: Int)(formula: Formula): Formula
    Eventually true in maximally n steps.

def eventuallyLt(n: Int)(formula: Formula): Formula
    Eventually true in less than n steps.

def factExists(pred: PartialFunction[Fact, Boolean]): Boolean
    Tests whether a fact exists in the fact database, which satisfies a predicate.

def getMonitorResult: MonitorResult[Event]
    Returns the result of a trace analysis for this monitor.

def getMonitors: List[Monitor[Event]]
    Returns the sub-monitors of a monitor.

def globally(formula: Formula): Formula
    Globally true (an LTL formula).

def hot(m: Int, n: Int)(block: PartialFunction[Event, Formula]): Formula
    A hot state waiting for an event to eventually match a transition (required) between m and n steps.

def hot(block: PartialFunction[Event, Formula]): Formula
    A hot state waiting for an event to eventually match a transition (required). The state remains active until the incoming event e matches the block, that is, until block.isDefinedAt(e) == true, in which case the state formula evaluates to block(e).

    At the end of the trace a hot state formula evaluates to False.

    As an example, consider the following monitor, which checks the property: "a command x eventually should be followed by a success".

    ```scala
    class Requirement extends Monitor[Event] {
        require {
            case COMMAND(x) =>
                hot {
                    case SUCCESS(`x`) => ok
                }
        }
    }
    ```

    **block**
    partial function representing the transitions leading out of the state.

    **returns**
    the hot state formula.

    **definition classes**: Formulas

def informal(name: Symbol)(explanation: String): Unit
    Used to enter explanations of properties in informal language.

def informal(explanation: String): Unit
    Used to enter explanations of properties in informal language.

def matches(predicate: PartialFunction[Event, Boolean]): Formula
    Matches current event against a predicate.

def monitor(monitors: Monitor[Event]*): Unit
    Adds monitors as sub-monitors to the current monitor.

def never(formula: Formula): Formula
    Never true (an LTL-inspired formula).
command verification in LADEE mission

class R42 extends Monitor[Event] {
  always {
    case COMMAND("ACS_MODE", _, time1, _) =>
      state {
        case COMMAND("ACS", _, time2, _) =>
          (time1, time2) beyond (1 second)
      }
  }
}

verified command sequence
implementation – formulas

```java
abstract class Formula {
    def apply(event: Event): Formula
    def reduce(): Formula = this
    def and(that: Formula): Formula = And(this, that).reduce()
    def until(that: Formula): Formula = Until(this, that).reduce()
    ...
}
```
case class State(block: Block) extends Formula {
  override def apply(event: Event): Formula =
    if (block.isDefinedAt(event)) block(event) else this
}

case class Step(block: Block) extends Formula {
  override def apply(event: Event): Formula =
    if (block.isDefinedAt(event)) block(event) else True
}

case class Strong(block: Block) extends Formula {
  override def apply(event: Event): Formula =
    if (block.isDefinedAt(event)) block(event) else False
}
globally and eventually

```scala
case class Globally(formula: Formula) extends Formula {
  override def apply(event: Event): Formula =
    And(formula(event), this).reduce()
}

case class Eventually(formula: Formula) extends Formula {
  override def apply(event: Event): Formula =
    Or(formula(event), this).reduce()
}
```
case class And(formula1: Formula, formula2: Formula) extends Formula {
  override def apply(event: Event): Formula =
    And(formula1(event), formula2(event)).reduce()

  override def reduce(): Formula = {
    (formula1, formula2) match {
      case (False, _) => False
      case (_, False) => False
      case (True, _) => formula2
      case (_, True) => formula1
      case (f1, f2) if f1 == f2 => f1
      case _ => this
    }
  }
}
def end(formula: Formula): Boolean =
  formula match {
    case State(_) => true
    case Hot(_ )  => false

    case Strong(_) => false
    case Weak(_ )  => true

    case Step(_)   => true

    case Globally(_) => true
    case Eventually(_) => false

    case And(formula1, formula2) => end(formula1) && end(formula2)
  }
future plans

• optimization
  – internal DSL is not analyzable
  – indexing: map incoming events to monitors

• application within LADEE mission
  – feature refinement (expressiveness)

• trace analysis in a broader perspective:
  – trace monitoring for embedded systems
  – trace mining
  – trace visualization

understanding complex systems by analyzing their execution