Formal Methods Efforts at JPL

IFIP WG 1.9/2.15 – Leuven, May 11, 2017

Klaus Havelund

With:
Rajeev Joshi (JPL)
Doron Peled (Bar Ilan University, Israel)
Sean Kauffman (University of Waterloo, Canada)
Current Work at JPL

• Scala for modeling

• Runtime verification with BDDs

• Event stream abstraction
Scala for Modeling
Using Scala for development of Hierarchical State Machines

With Rajeev Joshi
class Plant

instance variables

alarms : set of Alarm;
schedule : map Period to set of Expert;
inv PlantInv(alarms, schedule);

operations

PlantInv : set of Alarm * map Period to set of Expert ==>
bool
PlantInv(as, sch) ==
  return
  (forall p in set dom sch & sch(p) <> []) and
  (forall a in set as &
    forall p in set dom sch &
      exists expert in set sch(p) &
        a.GetReqQuali() in set expert.GetQuali());

types

public Period = token;

operations

public ExpertToPage: Alarm * Period ==>
Expert
ExpertToPage(a, p) ==
  let expert in set schedule(p) be st
    a.GetReqQuali() in set expert.GetQuali()
  in
    return expert
  pre a in set alarms and
    p in set dom schedule
  post let expert = RESULT
    in
      expert in set schedule(p) and
      a.GetReqQuali() in set expert.GetQuali();

VDM++

Chemical Plant Alarm System

Scala

class Plant(alarms: Set[Alarm],
    schedule: Map[Period, Set[Expert]]) {
    assert(PlantInv(alarms, schedule))

def PlantInv(alarms: Set[Alarm],
    schedule: Map[Period, Set[Expert]]): Boolean =
    (schedule.keySet forall { schedule(_,) != Set() }) &&
    (alarms forall { a =>
      schedule.keySet forall { p =>
        schedule(p) exists { expert =>
          a.reqQuali ? expert.quali
t
        }
      }
    })

    def ExpertToPage(a: Alarm, p: Period): Expert = {
    require(a ? alarms && p ? schedule.keySet)
    schedule(p) suchthat {expert =>
      a.reqQuali ? expert.quali}
    } ensuring { expert =>
      a.reqQuali ? expert.quali &&
      expert ? schedule(p) }
Modeling @ JPL

SysML
Languages: graphical, formal, programming

Graphical:
- UML
- SysML
- ...

Formal
- VDM
- Z
- PVS
- Promela
- ...

Programming
- Python
- Java,
- Scala,
- Ocaml
- ...

Languages:
- graphical
- formal
- programming

- UML
- SysML
- ...

- VDM
- Z
- PVS
- Promela
- ...

- Python
- Java,
- Scala,
- Ocaml
- ...

- graphical
- formal
- programming
Requirement

The number of loans that a book is part of should be less than or equal to the number of copies of the book.
class Book {
    name : String
    copies : Int
    library : Library

    loans : Set[Loan] =
    Set{ loan | loan : Loan ∧
        loan ∈ library.loans ∧ loan.book = this
    }

    fun isAvailable : Bool {
        size(loans) < copies
    }

    req CopiesPositive : copies > 0
    req SufficientCopies : size(loans) <= copies
}
trait Book extends Model {
  var name: String
  var copies: Int
  var library: Library

  def loans: Set[Loan] =
    library.loans.filter(_.book eq this)

  def isAvailable(): Boolean = loans.size < copies

  invariant("CopiesPositive") { copies > 0 }

  invariant("SufficientCopies") {
    loans.size <= copies
  }
}
Formal Verification of Scala Programs

Peter Muller and Alex Summers
Motivation for focusing on Hierarchical State Machines

- **HSMs used extensively** in FSW (on MSL, SMAP, M2020, FswCore)

- FSW typically has many HSMs interacting with each other and with devices

- During design, **hard to analyze** and understand complex interactions among HSMs, and how behavior is affected
  - by faults
  - by devices in off-nominal fashion (e.g., taking longer to warm up)

- **We need** a way to perform easy design and analysis of HSMs:
  - easy to **model** existing MSL/SMAP designs in expressive DSL
  - easy to write scenario **test cases**
  - easy to write properties to be **monitored** during tests
  - eventually **verify**
Approach

• Develop domain-specific language as an API (internal DSL) in the high-level strongly typed object-oriented and functional programming language Scala.

• This allows fast and safe prototyping of new functionality.

• Design and implement DSL for Hierarchical State Machines (HSMs) in Scala.

• Design and implement DSL for monitoring and testing in Scala.

• Develop automated testing framework.

• Develop formal verification technology for Scala.
Case Study: CBM (Coordinated Communications Behaviors), used on MSL/M2020/Europa Clipper

- CBM HSM inherited from MSL has ~50 states.
- Interacts with many devices and low-level subsystems in order to establish spacecraft state for telecom configuration and for communication windows.
- Complex interactions have led to surprises during MSL
  - Sol 696-698 anomaly resulted in 3 days of lost science due to frequency sweep not happening on time due to prolonged mounting of file system.
- Ideally, these should have been discovered during V&V - but hard to enumerate all interactions using traditional testing.
- Our approach should allow automatic test-case generation from high-level HSM and scenario descriptions; during each test case run, a set of monitored properties are checked automatically.
A Hierarchical State Machine

prep
entry { disable_subqueues("request", "abort") }
exit { enable_subqueues("abort") }

xband_prep
entry { Hga ! START_TRACK ; Self ! STEP }

STEP
xband_cfg
entry { config = load_config() ; Self ! STEP }
STEP / Sdst ! TURN_ON

DONE
active
entry { enable_subqueue("abort") }

dur1
entry { Timer ! START(5) }
TIMEOUT

dur2
entry { Timer ! START(300) }
TIMEOUT

dur3
entry { Timer ! START(5) }

INIT
STOP / Timer ! CANCEL

cleanup
entry { Hga ! STOP.Track ; Sdst ! TURN_OFF }
exit { enable_subqueue("request") }

DONE
idle
case object STEP extends CbmMessage("transition")
case object DONE extends CbmMessage("transition")
case object TIMEOUT extends CbmMessage("transition")
case object STOP extends CbmMessage("abort")
case object INIT extends CbmMessage("request")

class CbmHsm extends MslHsm {
  object top extends state() {}
  object idle extends state(top, true) {
    when {
      case INIT ⇒ in_window
    }
  }
  object in_window extends state(top) {}
  object prep extends state(in_window, true) {
    entry { disable_subqueues("request", "abort") }
    exit { enable_subqueues("abort") }
  }
  object xband_prep extends state(prep, true) {
    entry { Hga ! START_TRACK ; Self ! STEP }
    when { case STEP ⇒ xband_cfg }
  }
  object xband_cfg extends state(prep) {
    entry { config = load_config(); Self ! STEP }
    when {
      case STEP ⇒ stay exec { Sdst ! TURN_ON }
      case DONE ⇒ active
    }
  }
  object active extends state(in_window) {
    entry { enable_subqueues("abort") }
    when { case STOP ⇒ cleanup exec { Timer ! CANCEL } }
  }
...
}
// In state in_window, the request subqueue is disabled

class QueueCheck extends MSLMonitor {
    invariant("requestDisabled") {
        Cbm.inState("in_window") ⇒ !Cbm.isEnabled("request")
    }
}

// A Timer is never started while a previous timer is active

class TimerCheck extends MSLMonitor {
    always {
        case TIM_EVR_STARTED(_) ⇒ watch {
            case TIM_EVR_FIRED(_) | TIM_EVR_CANCELED(_) ⇒ ok
            case TIM_EVR_STARTED(_) ⇒ error("Timer restarted")
        }
    }
}
Plans for Test Case Generation

```scala
class scenario1 extends TestScenario {
  at(30) exec { Cbm ! INIT }
}

class scenario2 extends scenario1 {
  Cbm in "dur2" exec { Cbm ! STOP }
}
```

```scala
1 taskSet := set of tasks in scenario
2 while (taskSet ≠ ∅) {
3   choose T in taskSet with least value of T.time such that T is enabled
4   advance current time to T.time
5   perform T.action
6   while (there is an HSM that has a pending message) {
7     choose HSM H such that H has a pending message
8     execute H on its highest priority message
9   }
10 }
```
HSMs in Scala

```scala
trait HSM[Event] {...}

The HSM trait defines the following types and values used throughout:

```scala
type Code = Unit ⇒ Unit
type Target = (state, Code)
type Transitions = PartialFunction[Event, Target]
val noTransitions: Transitions = {case _ if false ⇒ null}
val skip: Code = (x: Unit) ⇒ {/
```
HSMs in Scala

case class state(parent: state = null, init: Boolean = false) {
  var entryCode: Unit ⇒ Unit = skip
  var exitCode: Unit ⇒ Unit = skip
  var transitions: Transitions = noTransitions

  ...

def entry(code: ⇒ Unit): Unit = {entryCode = (x: Unit) ⇒ code}
def exit(code: ⇒ Unit): Unit = {exitCode = (x: Unit) ⇒ code}
def when(ts: Transitions): Unit = {transitions = ts}

  implicit def state2Target(s: state): Target = (s, skip)
  implicit def state2Exec(s: state) = new {
    def exec(code: ⇒ Unit) = (s, (x: Unit) ⇒ code)
  }
}
HSMs in Scala

```scala
def initial (s: state): Unit = {current = s.getInnerMostState}

var initialState : state = null
if (parent != null && init) {parent. initialState = this}
def getInnerMostState: state =
  if (initialState == null) this else initialState .getInnerMostState
def getSuperStates: List [state] =
  (if (parent == null) Nil else parent.getSuperStates) ++List(this)
```
HSMs in Scala

var current: state = null

def submit(event: Event): Unit = {
    findTriggerHappyState(current, event) match {
      case None ⇒
      case Some(triggerState) ⇒
        val (transitionState, transitionCode) = triggerState.transitions(event)
        val targetState = transitionState.getInnerMostState
        val (exitStates, enterStates) = getExitEnter((current, targetState))
        for (s ← exitStates) s.exitCode()
        transitionCode()
        for (s ← enterStates) s.entryCode()
        current = targetState
    }
}
HSMs in Scala

```scala
def findTriggerHappyState(s: state, event: Event): Option[state] =
  if (s.transitions.isDefinedAt(event)) Some(s) else
  if (s.parent == null) None else findTriggerHappyState(s.parent, event)
```
Monitors in Scala

class Monitor[Event] {
  type Transitions = PartialFunction[Event, Set[state]]
  var states : Set[state] = Set()
  var invariants : List[(String, Unit ⇒ Boolean)] = Nil

  trait state { thisState ⇒
    var transitions : Transitions = noTransitions

    def apply(event:Event): Option[Set[state]] =
      if ( transitions.isDefinedAt(event)) Some(transitions(event)) else None
    def watch(ts: Transitions) { transitions = ts}
    def always(ts: Transitions) { transitions = ts andThen (- + this)}
  }

  def watch(ts: Transitions) = new state { watch(ts) }
  def always(ts: Transitions) = new state { always(ts) }
Publications


QTL
Quantified temporal logic for monitoring using BDDs

With Doron Peled (Bar Ilan University, Israel)
The Problem

logged in = \{John, Sandra\}

Property:

Whenever a user accesses a file:
• the user must have logged in
• the file must have been opened

logged in = \{John, Sandra\}
files open = \{tel\}
The Logic

\[ \phi ::= true \mid p(t_1, \ldots, t_n) \mid \neg \phi \mid \phi_1 \lor \phi_2 \mid \exists x \bullet \phi \mid @ \phi \mid \phi_1 S \phi_2 \]
\[ t ::= c \mid x \]

Derived Constructs

\[ false = \neg true \]
\[ \phi_1 \land \phi_2 = \neg (\neg \phi_1 \lor \neg \phi_2) \]
\[ \phi_1 \Rightarrow \phi_2 = \neg \phi_1 \lor \phi_2 \]
\[ \forall x \bullet \phi = \neg \exists x \bullet \neg \phi \]
\[ P \phi = true S \phi \]
\[ H \phi = \neg P \neg \phi \]
\[ [\phi_1, \phi_2] = (\neg \phi_2) S \phi_1 \]

Example

\[ \forall user \bullet \forall file \bullet \]
\[ access(user, file) \Rightarrow \]
\[ [login(user), logout(user)] \land \]
\[ [open(file), close(file)] \]
First Semantics

- \((\varepsilon, \sigma, i) \models true\).
- \((\varepsilon, \sigma, i) \models p(a)\) iff \(p(a) \in \sigma[i]\).
- \(([v \mapsto a], \sigma, i) \models p(v)\) if \(p(a) \in \sigma[i]\).
- \((\gamma, \sigma, i) \models (\phi \land \psi)\) if \((\gamma|_{\text{vars}(\phi)}, \sigma, i) \models \phi\) and \((\gamma|_{\text{vars}(\psi)}, \sigma, i) \models \psi\).
- \((\gamma, \sigma, i) \models \neg \phi\) if not \((\gamma, \sigma, i) \models \phi\).
- \((\gamma, \sigma, i) \models (\phi S \psi)\) if for some \(j \leq i\), \((\gamma|_{\text{vars}(\psi)}, \sigma, j) \models \psi\) and for all \(j < k \leq i\), \((\gamma|_{\text{vars}(\phi)}, \sigma, k) \models \phi\).
- \((\gamma, \sigma, i) \models \bigoplus \phi\) if \(i > 1\) and \((\gamma, \sigma, i - 1) \models \phi\).
- \((\gamma, \sigma, i) \models \exists x \phi\) if there exists \(a \in \text{domain}(x)\) such that \(^5\) \((\gamma[x \mapsto a], \sigma, i) \models \phi\).

\[(\phi S \psi) = (\psi \lor (\phi \land \bigoplus \phi S \psi))\]

- \((\gamma, \sigma, i) \models (\phi S \psi)\) if \((\gamma|_{\text{vars}(\psi)}, \sigma, i) \models \psi\) or both \((\gamma|_{\text{vars}(\phi)}, \sigma, i) \models \phi\) and \((\gamma, \sigma, i - 1) \models (\phi S \psi)\).
Second Semantics

- $I[\varphi, \sigma, 0] = \emptyset$.
- $I[\text{true}, \sigma, i] = \{\varepsilon\}$.
- $I[p(a), \sigma, i] = \text{if } p(a) \in \sigma[i] \text{ then } \varepsilon, \text{ else } \emptyset$.
- $I[p(v), \sigma, i] = \{[v \mapsto a] | p(a) \in \sigma[i]\}$.
- $I[(\varphi \land \psi), \sigma, i] = I[\varphi, \sigma, i] \cap I[\psi, \sigma, i]$.
- $I[\neg \varphi, \sigma, i] = A_{\text{vars}(\varphi)} \setminus I[\varphi, \sigma, i]$.
- $I[(\varphi \lor \psi), \sigma, i] = I[\psi, \sigma, i] \cup (I[\varphi, \sigma, i] \cap I[(\varphi \land \psi), \sigma, i - 1])$.
- $I[\forall x \varphi, \sigma, i] = I[\varphi, \sigma, i - 1]$.
- $I[\exists x \varphi, \sigma, i] = \text{proj}(I[\varphi, \sigma, i], \{x\})$. 
Mapping data to BDDs

\[ \text{prop} \ p : \ \forall f . \ \text{close}(f) \rightarrow \exists m . \ P \ \text{open}(f,m) \]

\[ \langle \text{open}(\text{input, read}), \text{open}(\text{output, write}), \text{close}(\text{out}) \rangle \]

\[ f \mapsto \langle \text{input} \mapsto 000, \text{output} \mapsto 001, \text{out} \mapsto 010 \rangle, \]
\[ m \mapsto \langle \text{read} \mapsto 000, \text{write} \mapsto 001 \rangle \]
Algorithm

1. Initially, for each subformula $\varphi$, $\text{now}(\varphi) = 0$.
2. Observe a new state (as set of ground predicates) $s$ as input.
3. Let $\text{pre} := \text{now}$.
4. Make the following updates for each subformula. If $\varphi$ is a subformula of $\psi$ then $\text{now}(\varphi)$ is updated before $\text{now}(\psi)$.
   - $\text{now}(\text{false}) = 0$
   - $\text{now}(p(a)) = \text{if } p(a) \in s \text{ then } 1 \text{ else } 0$
   - $\text{now}(p(x)) = \text{if } p(a) \in s \text{ then } \text{build}(x,a) \text{ else } 0$
   - $\text{now}(\varphi \land \psi) = \text{and}(\text{now}(\varphi), \text{now}(\psi))$
   - $\text{now}(\neg \varphi) = \text{not}(\text{now}(\varphi))$
   - $\text{now}(\varphi \lor \psi) = \text{or}(\text{now}(\psi), \text{and}(\text{now}(\varphi), \text{pre}(\varphi \lor \psi)))$
   - $\text{now}(\ominus \varphi) = \text{pre}(\varphi)$
   - $\text{now}(\exists x \varphi) = \text{exists}(\langle x_0, \ldots , x_n \rangle, \text{now}(\varphi))$
5. Goto step 2.
The Implementation
Binary Decision Diagrams

logged in = \{\text{John, Sandra}\}

Users as bit patterns

The set \{\text{John, Sandra}\} as a Boolean expression:

\[(b_1 = 0 \land b_2 = 0) \lor (b_1 = 1 \land b_2 = 1)\]

login(“John”) login(“Sandra”) now

Boolean expression as BDD:
The Structure of a Property

\[ \text{prop secure : } \forall \text{(user)} \forall \text{(file)} \text{ access(user, file) } \rightarrow \left[ \text{login(user)}, \text{logout(user)} \right] \& \left[ \text{open(file)}, \text{close(file)} \right] \]
Example Application

\textbf{prop secure :}

\texttt{forall (user) forall (file) access(user,file) \to [login(user),logout(user))}
\&
\texttt{[open(file),close(file))}

```
class Formula_secure extends Formula {
  def evaluate(): Boolean = {
    var now = Nil
    var user
    var file
    var_var_user :: var_var_file :: Nil = declareVariables("user", "file")

    val now(0) = build("close")("file")
    val now(9) = build("open")("file")
    val now(8) = now(9).or(now(10).not().and(pre(8)))
    val now(7) = build("logout")("user")
    val now(6) = build("login")("user")
    val now(5) = now(6).or(now(7).not().and(pre(5)))
    val now(4) = now(5).and(now(8))
    val now(3) = build("access")("user", "file")
    val now(2) = now(3).not().or(now(4))
    val now(1) = now(2).forAll(var_file)
    val now(0) = now(1).forAll(var_user)

    pre = Array.fill(11)(False)
    now = Array.fill(11)(False)

    tmp = now
    now = pre
    pre = tmp
    !tmp(0).isZero
  }

  override def evaluate(): Boolean = {
    now(10) = build("close")("file")
    now(9) = build("open")("file")
    now(8) = now(9).or(now(10).not().and(pre(8)))
    now(7) = build("logout")("user")
    now(6) = build("login")("user")
    now(5) = now(6).or(now(7).not().and(pre(5)))
    now(4) = now(5).and(now(8))
    now(3) = build("access")("user", "file")
    now(2) = now(3).not().or(now(4))
    now(1) = now(2).forAll(var_file)
    now(0) = now(1).forAll(var_user)

    pre = Array.fill(11)(False)
    now = Array.fill(11)(False)
  }
```
prop p: \( \forall f . \, \text{close}(f) \rightarrow \exists m . \, P \text{open}(f,m) \)

class Formula_p extends Formula {
  var pre: Array[BDD] = Array.fill(6)(False)
  var now: Array[BDD] = Array.fill(6)(False)
  var tmp: Array[BDD] = null
  val var_f :: var_m :: Nil =
    declareVariables("f", "m")

  override def evaluate(): Boolean = {
    now(5) = build("open")((V("f"), V("m")))
    now(4) = now(5).or(pre(4))
    now(3) = now(4).exist(var_m)
    now(2) = build("close")((V("f")))
    now(1) = now(2).not().or(now(3))
    now(0) = now(1).forall(var_f)
    tmp = now; now = pre; pre = tmp
    !tmp(0).isZero
  }
}
Evaluation of Trace

\[ \langle \text{open(input, read)}, \text{open(output, write)}, \text{close(out)} \rangle \]
Fig. 2: BDD for nodes for (left) subformula 5 on first event, (mid) subformula 5 on second event, (right) subformula 4 on second event
Evaluation Properties in QTL

\[
\text{prop access : forall } u . \text{ forall } f \ . \\
\quad \text{access}(u,f) \rightarrow [\text{login}(u),\text{logout}(u)) \& [\text{open}(f),\text{close}(f))
\]

\[
\text{prop file : forall } f . \\
\quad \text{close}(f) \rightarrow \text{exists } m . \@ [\text{open}(f,m),\text{close}(f))
\]

\[
\text{prop fifo : forall } x . \\
\quad (\text{enter}(x) \rightarrow ! @ P \text{ enter}(x)) \& \\
\quad (\text{exit}(x) \rightarrow ! @ P \text{ exit}(x)) \& \\
\quad (\text{exit}(x) \rightarrow @ P \text{ enter}(x)) \& \\
\quad (\text{forall } y . (\text{exit}(y) \& P (\text{enter}(y) \& @ P \text{ enter}(x)))) \rightarrow @ P \text{ exit}(x))
\]
Evaluation Properties in MonPoly

/* access */  FORALL u. (FORALL f.
    ( access(u,f) IMPLIES
        ((( NOT logout(u)) SINCE login(u)) AND (NOT close(f) SINCE[0,*] open(f)))))

/* file */  FORALL f.
    (close(f) IMPLIES (EXISTS m . PREVIOUS ( NOT close(f) SINCE[0,*] open(f,m))))

/* fifo */  FORALL x. ( 
    (enter(x) IMPLIES NOT PREVIOUS ONCE[0,*] enter(x)) AND
    (exit (x) IMPLIES NOT PREVIOUS ONCE[0,*] exit(x)) AND
    (exit (x) IMPLIES PREVIOUS ONCE[0,*] enter(x)) AND
    FORALL y.
        (( exit (y) AND ONCE[0,*] (enter(y) AND PREVIOUS ONCE[0,*] enter(x)))
        IMPLIES PREVIOUS ONCE exit(x)))
### Evaluation Results

#### Table 1: Evaluation of QTL and MONPOLY

<table>
<thead>
<tr>
<th>Property</th>
<th>Trace length</th>
<th>MONPOLY (sec)</th>
<th>QTL (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>bits per var.: 20 (40, 60)</td>
</tr>
<tr>
<td>ACCESS</td>
<td>11,006</td>
<td>1.9</td>
<td>3.1 (3.3, 3.2)</td>
</tr>
<tr>
<td></td>
<td>110,006</td>
<td>241.9</td>
<td>6.1 (9.1, 10.9)</td>
</tr>
<tr>
<td></td>
<td>1,100,006</td>
<td>58,455.8</td>
<td>36.8 (61.9, 88.8)</td>
</tr>
<tr>
<td>FILE</td>
<td>11,004</td>
<td>61.1</td>
<td>2.8 (2.8, 3.0)</td>
</tr>
<tr>
<td></td>
<td>110,004</td>
<td>7,348.7</td>
<td>6.3 (6.5, 8.6)</td>
</tr>
<tr>
<td></td>
<td>1,100,004</td>
<td>DNF</td>
<td>30.3 (43.9, 59.5)</td>
</tr>
<tr>
<td>FIFO</td>
<td>5,051</td>
<td>158.3</td>
<td>195.4 (OOM, ?)</td>
</tr>
<tr>
<td></td>
<td>10,101</td>
<td>1140.0</td>
<td>ERR (?, ?)</td>
</tr>
</tbody>
</table>
Nfer
A tool for event stream abstraction

With Rajeev Joshi and Sean Kauffman (U. of Waterloo, Canada)
EVR(BOOT_S, 42, 3)
Current Situation

vrf1.py
tst2.awk
chk3.py
Nfer
A Tool for Event Abstraction

visualize & verify

Abstract using DSL
Grammar for DSL

specification ::= rule+ | module+
module ::= 'module' identifier '{' [imports] rule* '}'
imports ::= 'import' identifier (',' identifier)* ';
rule ::= identifier ':-' intervalExpression [whereExpression] [mapExpression] [endPoints]
intervalExpression ::= primaryIntervalExpression (intervalOp primaryIntervalExpression)*
primaryIntervalExpression ::= atomicIntervalExpression | parenIntervalExpression
atomicIntervalExpression ::= [label] identifier
parenIntervalExpression ::= '( ' intervalExpression ' )'
label ::= identifier ':'
whereExpression ::= 'where' expression
mapExpression ::= 'map' '{' identifier '->' expression (',' identifier '->' expression)* '}'
endPoints ::= 'begin' expression 'end' expression
intervalOp ::= 'also' | 'before' | 'meet' | 'during' | 'start' | 'finish' | 'overlap' | 'slice' | 'coincide'
expression ::= comparisonExpression (andorOp comparisonExpression)*
comparisonExpression ::= plusminusExpression (comparisonOp plusminusExpression)*
plusminusExpression ::= muldivExpression (plusminusOp muldivExpression)*
muldivExpression ::= unaryExpression (muldivOp unaryExpression)*
unaryExpression ::= stringLiteral | intLiteral | doubleLiteral | booleanLiteral |
                  beginTime | endTime | mapField | callExpression |
                  '(' expression ')') | unaryOp unaryExpression
mapField ::= identifier '.' identifier
beginTime ::= identifier '.' BEGIN
endTime ::= identifier '.' END
unaryOp ::= '-' | '!' 
muldivOp ::= '*' | '/' | '%' 
plusminusOp ::= '+' | '-'
comparisonOp ::= '<' | '<=' | '>' | '>=' | '=' | '!='
andorOp ::= '&&' | '||'
callExpression ::= 'call' '(' identifier ')'
Event Abstraction with nfer

<table>
<thead>
<tr>
<th>NAME</th>
<th>TIME</th>
<th>PARAMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOWNLINK</td>
<td>10</td>
<td>size -&gt; 430</td>
</tr>
<tr>
<td>BOOT_S</td>
<td>42</td>
<td>count -&gt; 3</td>
</tr>
<tr>
<td>TURN_ANTENNA</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>START_RADIO</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>DBOOT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOWNLINK</td>
<td>100</td>
<td>size -&gt; 420</td>
</tr>
<tr>
<td>BOOT_E</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>BOOT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOP_RADIO</td>
<td>205</td>
<td></td>
</tr>
<tr>
<td>DBOOT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOT_S</td>
<td>255</td>
<td>count -&gt; 4</td>
</tr>
<tr>
<td>START_RADIO</td>
<td>286</td>
<td></td>
</tr>
<tr>
<td>BOOT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOT_E</td>
<td>312</td>
<td></td>
</tr>
<tr>
<td>RISK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TURN_ANTENNA</td>
<td>412</td>
<td></td>
</tr>
</tbody>
</table>
Based on Allen Logic

- A ← B before C
- A ← B during C
- A ← B start C
- A ← B overlap C
- A ← B meet C
- A ← B coincide C
- A ← B finish C
- A ← B slice C
Specification in new DSL

BOOT :- BOOT_S before BOOT_E
    map {count -> BOOT_S.count}

DBOOT :- b1:BOOT before b2:BOOT
    where b2.begin - b1.end <= 300
    map {count -> b2.count}

RISK :- DOWNLINK during DBOOT
    map {count -> DBOOT.count}
module Booting {
    BOOT :- BOOT_S before BOOT_E
        map {count -> BOOT_S.count}
}

module DoubleBooting {
    import Booting;

    DBOOT :- b1:BOOT before b2:BOOT
        where b2.begin - b1.end <= 300
        map {count -> b2.count}
}

module Risking {
    import DoubleBooting;

    RISK :- DOWNLINK during DBOOT
        map {count -> DBOOT.count}
}
nfer Scala DSL

```scala
class DoubleBoot extends Nfer {
  "BOOT" :— ("BOOT_S" before "BOOT_E" map {
    case (m1,m2) ⇒ Map("count" → m1("count"))
  })

  "DBOOT" :— ("BOOT" before "BOOT" within 300 map (_._2))

  "RISK" :— ("DOWNLINK" during "DBOOT" map (_._2))
}
```
BOOT :- BOOT_S before BOOT_E
  where BOOT_E.begin - BOOT_S.end <= 5000
  map {count -> BOOT_S.count}

DBOOT :- b1:BOOT before b2:BOOT
  where b2.begin - b1.end <= 10000
  map {count -> b1.count}

RISK :- DOWNLINK during DBOOT
  map {count -> DBOOT.count}
package nfer

nfer is a Scala package for generating abstractions from event traces.

Introduction

nfer allows to define intervals over an event trace. The intervals indicate abstractions that for example can be visualized. An interval consists of a name, two time stamps: the beginning and the end of the interval, and a map from variable names (strings) to values, indicating data that the interval carries.

Writing Specifications

As an example, consider a scenario where we want to be informed if a downlink operation occurs during a 5-minute time interval where the flight computer reboots twice. This scenario could cause a potential loss of downlink information. We want to identify the following intervals.

A BOOT represents an interval during which the rover software is rebooting.

A DBOOT (double boot) represents an interval during which the rover reboots twice within a 5-minute timeframe.

A RISK represents an interval during which the rover reboots twice, and during which also attempts to downlink information.

Our objective now is to formalize the definition of such intervals in the nfer specification language, and extract these intervals from a telemetry stream, based on such a specification. Specifically, in this case, we need a formalism for formally defining the following three kinds of intervals.

A BOOT interval starts with a BOOT_S (boot start) event and ends with a BOOT_E (boot end) event.

A DBOOT (double boot) interval consists of two consecutive BOOT intervals, with no more than 5-minutes from the end of the first BOOT interval to the start of the second BOOT interval.

A RISK interval is a DBOOT interval during which a DOWNLINK occurs.

The following time line illustrates a scenario with two BOOTS, hence a double boot, and a downlink occurring during this period.

```
----- | ------------- | ------------- | ------------- | ------
|       | BOOT_S     |       | BOOT_S     | BOOT_E |
|       |           |       | BOOT_S     |         |
| DOWNLINK
```

The intervals can be formalized with the following nfer specification:

```scala
BOOT :- BOOT_S before BOOT_E
   where BOOT_E.begin - BOOT_S.end <= 5000
   map {count -> BOOT_S.count}

DBOOT :- b1:BOOT before b2:BOOT
   where b2.begin - b1.end <= 10000
   map {count -> b1.count}
```
GUI for Entering Rules using a Log

Detailed design and programming by Nathaniel Guy
END