Monitoring with Data Automata

Klaus Havelund
Jet Propulsion Laboratory, USA

WG 1.9/2.15 Verified Software

July 14-16, 2014

Jet Propulsion Laboratory
California Institute of Technology

NASA/JPL Laboratory
for Reliable Software

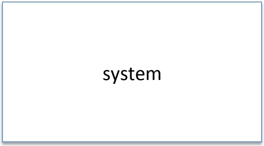
Definition of "Runtime Verification"

Definition (Runtime Verification)

Runtime Verification is the discipline of computer science dedicated to the analysis of system executions, including checking them against formalized specifications.

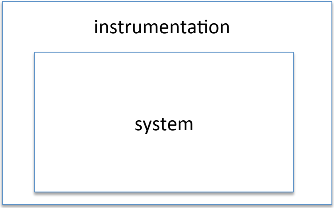
Runtime verification

- Start with a system to monitor.



Runtime verification

- *Instrument* the system to record relevant events.

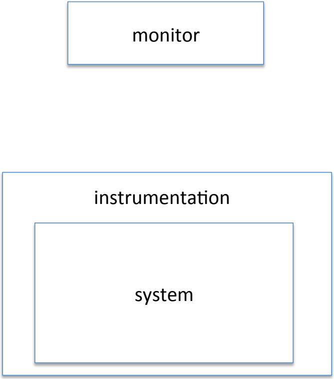


instrumentation

system

Runtime verification

- *Provide* a monitor.



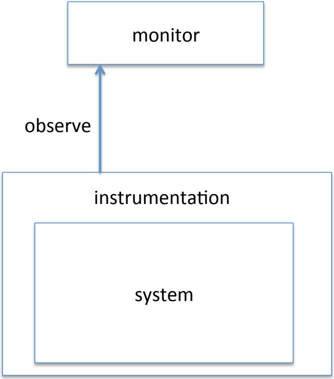
monitor

instrumentation

system

Runtime verification

- *Dispatch* each received event to the monitor.



monitor

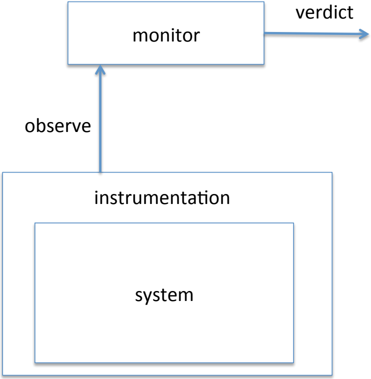
observe

instrumentation

system

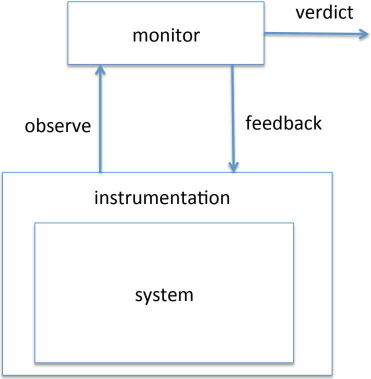
Runtime verification

- Compute a *verdict* for the trace received so far.



Runtime verification

- Possibly generate *feedback* to the system.


monitor

verdict

observe

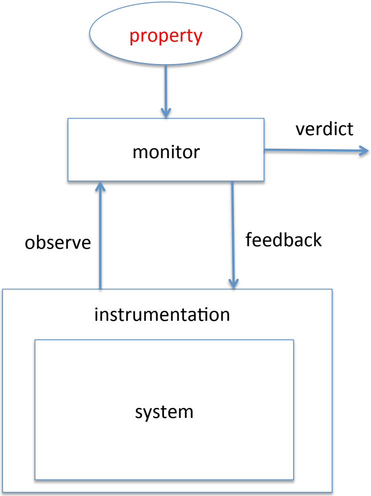
feedback

instrumentation

system

Runtime verification

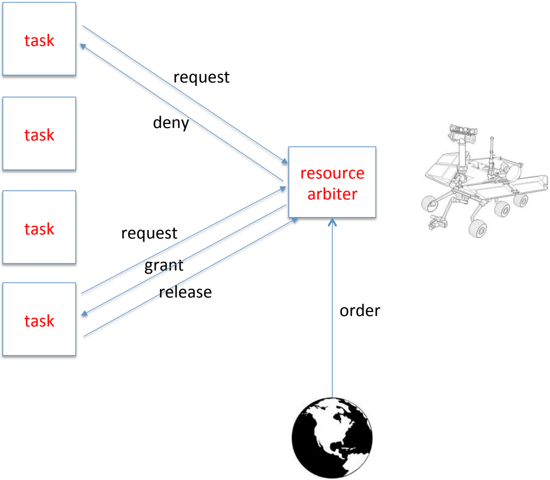
- We might possibly have synthesized monitor from a *property*.



Data Automata

(DAUT)

Granting and releasing of locks



task

request

deny

resource arbiter

task

task

request

grant

release

task

order

Resource allocation requirements

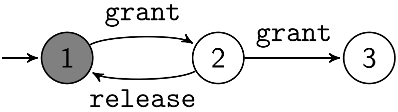
Requirement R_1

A grant of a resource to a task must be followed by a release of that resource by the same task, without another grant of that resource in between (to the same task or any other task).

A state machine

Requirement R_1

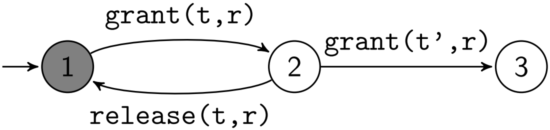
A grant of a resource to a task must be followed by a release of that resource by the same task, without another grant of that resource in between (to the same task or any other task).



A state machine with parameters

Requirement R_1

A grant of a resource to a task must be followed by a release of that resource by the same task, without another grant of that resource in between (to the same task or any other task).



Consider the following trace

$$grant(t_1, antenna)$$

$$grant(t_2, motor_2)$$

$$grant(t_3, motor_4)$$

Monitor configuration after these three events

$$\{S2(t_1, antenna), S2(t_2, motor_2), S2(t_3, motor_4)\}$$

Design of a DSL

Scala is a high-level unifying language

- Object-oriented + functional programming features
- Strongly typed with type inference
- Script-like, semicolon inference
- Sets, list, maps, iterators, comprehensions
- Lots of libraries
- Compiles to JVM
- Lively growing community

References

http://www.havelund.com

Monitoring with Data Automata Klaus Havelund. ISoLA 2014 – 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Track: Statistical Model Checking, Past Present and Future. Organized by: K. Larsen and A. Legay. Editors: T. Margaria and B. Steffen. Springer, LNCS. Corfu, Greece, October 8-11, 2014.

Data Automata in Scala Klaus Havelund. TASE 2014 - The 8th International Symposium on Theoretical Aspects of Software Engineering, IEEE proceedings. Changsha, China, September 1-3, 2014.

Rule-based runtime verification revisited Klaus Havelund. Software Tools for Technology Transfer (STTT). Springer. April, 2014.

TraceContract: A Scala DSL for Trace Analysis Howard Barringer and Klaus Havelund. FM 2011 - 17th International Symposium on Formal Methods. Springer, LNCS 6664. Limerick, Ireland, June 20-24, 2011.

Data Automata

- as an **external** DSL
    1. small language with focused functionality
    2. specialized parser programmed using parser generator

Data Automata

- as an **external** DSL
  1. small language with focused functionality
  2. specialized parser programmed using parser generator
  3. advantages:
     1. complete control over language syntax
     2. analyzable

Data Automata

- as an **external** DSL
    1. small language with focused functionality
    2. specialized parser programmed using parser generator
    3. advantages:
        1. complete control over language syntax
        2. analyzable
- as an **internal** DSL
    1. API in SCALA
    2. using SCALA's infra-structure (compiler, IDEs, ...)

Data Automata

- as an **external** DSL
  1. small language with focused functionality
  2. specialized parser programmed using parser generator
  3. advantages:
     1. complete control over language syntax
     2. analyzable
- as an **internal** DSL
  1. API in SCALA
  2. using SCALA's infra-structure (compiler, IDEs, ...)
  3. advantages:
     1. expressive, the programming language is never far away
     2. easier to develop/adapt (although, sometimes not)
     3. allows use of existing tools such as type checkers, IDEs, etc.

An external DSL

Resource allocation requirements

Requirement R_1

A grant of a resource to a task must be followed by a release of that resource by the same task, without another grant of that resource in between (to the same task or any other task).

Requirement R_2

A resource cannot be released by a task, which has not been granted the resource.

R_1 and R_2 as a state machine in DAUT

```
monitor R1R2 {
  init  always Start  {
    grant(t,  r) → Granted(t,r)
     release (t,  r)  ::  ¬Granted(t,r) → error
  }

  hot Granted(t,r)  {
     release (t,r)  → ok
    grant(_,r) → error
  }
}
```

top level abbreviation

```
monitor R1R2 {
  grant(t, r) → Granted(t,r)
  release (t, r) :: ¬Granted(t,r) → error

  hot Granted(t,r) {
    release (t,r) → ok
    grant(_,r) → error
  }
}
```

Requirement R_1

```
monitor R1 {
  grant(t,r) → hot {
    release (t,r) → ok
    grant(_,r) → error
  }
}
```

Syntax

```
⟨Specification⟩ ::=  ⟨Monitor⟩*

⟨Monitor⟩ ::=  monitor ⟨Id⟩ ‘{’ ⟨Transition⟩* ⟨State⟩* ‘}’

⟨State⟩ ::=  ⟨Modifier⟩* ⟨Id⟩ ⌈ (⟨Id⟩**) ⌉ ⌈ ‘{’ ⟨Transition⟩* ‘}’ ⌉

⟨Modifier⟩ ::=  init | hot | always

⟨Transition⟩ ::=  ⟨Pattern⟩ ‘::’ ⟨Condition⟩ ‘→’ ⟨Action⟩**

⟨Pattern⟩ ::=  ⟨Id⟩ ‘(’⟨Id⟩**‘)’

⟨Condition⟩ ::=  ⟨Condition⟩ ‘∧’ ⟨Condition⟩
        |  ⟨Condition⟩ ‘∨’ ⟨Condition⟩
        |  ‘¬’ ⟨Condition⟩
        |  ‘(’⟨Condition⟩‘)’
        |  ⟨Expression⟩ ⟨relop⟩ ⟨Expression⟩
        |  ⟨Id⟩ ⌈ ‘(’⟨Expression⟩**‘)’ ⌉

⟨Action⟩ ::=  ok
        |  error
        |  ⟨Id⟩ ⌈ ‘(’⟨Expression⟩**‘)’ ⌉
        |  if ‘(’ ⟨Condition⟩ ‘)’ then ⟨Action⟩ else ⟨Action⟩
        |  ⟨Modifier⟩* ‘{’ ⟨Transition⟩* ‘}’
```

Semantics part 1/3

$$\boxed{\text{E}}\ \frac{con, con \overset{e}{\hookrightarrow} b, con'}{con \xrightarrow{e,b} con'}$$

$$\boxed{\text{E-ss}_1}\ \frac{}{con, \{\} \overset{e}{\hookrightarrow} (true, \{\})}$$

$$\boxed{\text{E-ss}_2}\ \frac{\begin{array}{c} con, s \overset{e}{\longmapsto} res \\ con, ss \overset{e}{\hookrightarrow} res' \end{array}}{con, s \cup ss \overset{e}{\hookrightarrow} res \oplus res'}$$

Semantics part 2/3

$$\boxed{\text{E-s}_1}\ \frac{con, s.env, s.ts \overset{e}{\Longrightarrow} \bot}{con, s \overset{e}{\longmapsto} true, \{s\}}$$

$$\boxed{\text{E-s}_2}\ \frac{con, s.env, s.ts \overset{e}{\Longrightarrow} res}{con, s \overset{e}{\longmapsto} res}$$

$$\boxed{\text{E-ts}_1}\ \frac{}{con, env, Nil \overset{e}{\Longrightarrow} \bot}$$

$$\boxed{\text{E-ts}_2}\ \frac{\begin{array}{c} con, env, t \overset{e}{\rightharpoondown} res_\bot \\ con, env, ts \overset{e}{\Longrightarrow} res'_\bot \end{array}}{con, env, \langle t \rangle {}^\frown ts \overset{e}{\Longrightarrow} res_\bot \oplus_\bot res'_\bot}$$

Semantics part 3/3

$$\boxed{\text{E-t}_1}\ \frac{\begin{array}{c} t~\textbf{is}~`pat :: cond \to rhs' \\ [\![pat]\!]^{P} env~~e = \bot \end{array}}{con, env, t \overset{e}{\rightharpoonup} \bot}$$

$$\boxed{\text{E-t}_2}\ \frac{\begin{array}{c} t~\textbf{is}~`pat :: cond \to rhs' \\ [\![pat]\!]^{P} env~~e = env' \\ [\![cond]\!]^{C} con~env' = false \end{array}}{con, env, t \overset{e}{\rightharpoonup} \bot}$$

$$\boxed{\text{E-t}_3}\ \frac{\begin{array}{c} t~\textbf{is}~`pat :: cond \to rhs' \\ [\![pat]\!]^{P} env~~e = env' \\ [\![cond]\!]^{C} con~env' = true \\ [\![rhs]\!]^{R} con~env' = res \end{array}}{con, env, t \overset{e}{\rightharpoonup} res}$$

Implementation of external DSL

Abstract syntax

```
case class  Specification (automata:  List [Automaton])
case class  Automaton(name: Id, states:  List [StateDef])

case class  StateDef(
   modifiers :  List [Modifier] ,
   name: Id,
   formals :  List [Id] ,
    transitions  :  List [ Transition ])

case class  Transition (
   pattern :  Pattern ,
   condition :  Option[Condition] ,
   rhs :  List [StateExp])

 trait  Pattern
case class  FormalEvent(name: Id,  formals :  List [Id])  extends Pattern
case object Any extends Pattern
```

Parser

```
object Grammar extends JavaTokenParsers {
  def specification : Parser[ Specification ] =
    rep(automaton) ^^ {
      case automata ⇒ transform( Specification (automata))
    }

  def automaton: Parser[Automaton] =
    "monitor" → ident ˜ ("{" → rep( transition ) ˜ rep( statedef ) ← "}") ^^
      {
        case name ˜ ( transitions ˜ statedefs ) ⇒
          if ( transitions .isEmpty)
            Automaton(name, statedefs)
          else { // derived form
            val initialState =
              StateDef( List ( init , always ), "StartFromHere", Nil, transitions )
            Automaton(name, initialState :: statedefs )
          }
      }
```

Interpreter interface

```
trait Monitor[Event] {
  def verify (event: Event)
  def end()
}
```

Interpreter

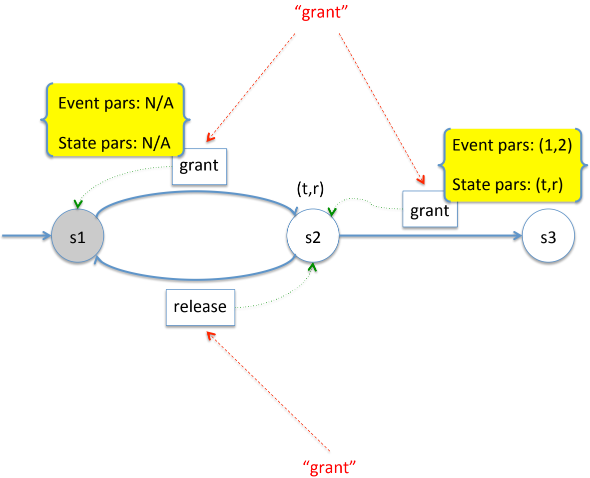
```
class MonitorImpl(automaton: Automaton) extends Monitor[Event] {
  case class State(name: Id, values: List[Value]) {
    var env: Env = null
  }

  type Config = Set[State]
  type Result = (Boolean, Config)

  var currentConfig: Config = initialConfig(automaton)

  def verify(event: Event) {
    val (status, con) = eval(currentConfig)(event)
    if (!status) println("*** error")
    currentConfig = con
  }
  ...
}
```

Indexing optimization



An internal DSL

Event type modeled in internal DSL

```
trait Event
case class grant(task: String, resource: String) extends Event
case class release(task: String, resource: String) extends Event
```

Properties modeled in internal DSL

```
class R1R2 extends Monitor[Event] {
  Always {
    case grant(t, r) ⇒ Granted(t, r)
    case release(t, r) if !Granted(t, r) ⇒ error
  }

  case class Granted(t: String, r: String) extends state{
    Watch {
      case release(‘t‘, ‘r‘) ⇒ ok
      case grant(_, ‘r‘) ⇒ error
    }
  }
}
```

Properties modeled in internal DSL

```
class R1 extends Monitor[Event] {
  Always {
    case grant(t, r) ⇒ hot {
      case release(‘t‘, ‘r‘) ⇒ ok
      case grant(_, ‘r‘) ⇒ error
    }
  }
}
```

Properties modeled in internal DSL

```
object Main {
  def main(args: Array[String]) {
    val obs = new R1R2

    obs. verify (grant("t1", "A"))
    obs. verify (grant("t2", "A"))
    obs. verify ( release ("t2", "A"))
    obs. verify ( release ("t1", "B"))
    obs.end()
  }
}
```

amazon.com

S. Hallé and R. Villemaire,
"Runtime enforcement of web service message contracts with data",
IEEE Transactions on Services Computing, vol. 5, no. 2, 2012. –
formalized in $\mathrm{LTL\text{-}FO}^{+}$.

XML based client server communication

client

XML

server

Example of XML message

```
<CartAdd>
  <CartId>1</CartId>
  <Items>
    <Item>
      <ASIN>10</ASIN>
    </Item>
    <Item>
      <ASIN>20</ASIN>
    </Item>
  </Items>
</CartAdd>
```

Amazon E-Commerce Service

ItemSearch(txt)	→	search items on site
CartCreate(its)	→	create cart with items
CartCreateResponse(c)	←	get cart id back
CartGetResponse(c, its)	←	result of get query
CartAdd(c, its)	→	add items
CartRemove(c, its)	→	remove items
CartClear(c)	→	clear cart
CartDelete(c)	→	delete cart

Definition of events

```
case class Item(asin : String )

 trait Event
case class ItemSearch(text : String ) extends Event
case class CartCreate(items: List [Item]) extends Event
case class CartCreateResponse(id: Int ) extends Event
case class CartGetResponse(id: Int , items: List [Item]) extends Event
case class CartAdd(id: Int , items: List [Item]) extends Event
case class CartRemove(id:Int, items: List [Item]) extends Event
case class CartClear( id : Int ) extends Event
case class CartDelete( id : Int ) extends Event
```

From XML to objects

```
def xmlToObject(xml:scala.xml.Node):Event =
  xml match {
    case x @ <CartAdd>{ _* }</CartAdd> ⇒
      CartAdd(getId(x), getItems(x))
     ...
  }

def xmlStringToObject(msg:String):Event = {
  val xml = scala.xml.XML.loadString(msg)
  xmlToObject(xml)
}

def getId(xml:scala.xml.Node):Int =
  (xml \ "CartId").text.toInt

def getItems(xml:scala.xml.Node):List[Item] =
  (xml \ "Items" \ "Item" \ "ASIN").
     toList.map(i ⇒ Item(i.text))
```

Properties

- **Property 1** - *Until a cart is created, the only operation allowed is ItemSearch.*
- **Property 2** - *A client cannot remove something from a cart that has just been emptied.*
- **Property 3** - *A client cannot add the same item twice to the shopping cart.*
- **Property 4** - *A shopping cart created with an item should contain that item until it is deleted.*
- **Property 5** - *A client cannot add items to a non-existing cart.*

Properties formalized

```
class Property1 extends Monitor[Event] {
  Unless {
    case ItemSearch(_) ⇒ ok
    case _ ⇒ error
  } {
    case CartCreate(_) ⇒ ok
  }
}

class Property2 extends Monitor[Event] {
  Always {
    case CartClear(c) ⇒ unless {
      case CartRemove(‘c‘, _) ⇒ error
    } {
      case CartAdd(‘c‘, _) ⇒ ok
    }
  }
}
```

```
class Property3 extends Monitor[Event] {
  Always {
    case CartCreate(items) ⇒ next {
      case CartCreateResponse(c) ⇒ always {
        case CartAdd(‘c‘, items_) ⇒ items disjointWith items_
      }
    }
  }
}

class Property4 extends Monitor[Event] {
  Always {
    case CartAdd(c, items) ⇒
      for (i ∈ items) yield unless {
        case CartGetResponse(‘c‘, items_) ⇒ items_ contains i
      } {
        case CartRemove(‘c‘, items_) if items_ contains i ⇒ ok
      }
  }
}
```

```
class Property5 extends Monitor[Event] {
  Always {
    case CartCreateResponse(c) ⇒ CartCreated(c)
    case CartAdd(c, _) if !CartCreated(c) ⇒ error
  }

  case class CartCreated(c: Int) extends state {
    Watch {
      case CartDelete(‘c‘) ⇒ ok
    }
  }
}
```

Recall property 3

- **Property 3** - *A client cannot add the same item twice to the shopping cart.*

Property 3 made less strict

```
class  Property3Liberalized  extends Monitor[Event] {
  Always {
    case CartCreate(items) ⇒ next {
      case CartCreateResponse(c) ⇒ CartCreated(c, items)
    }
  }

  case class CartCreated(id: Int, items: List[Item]) extends state {
    Watch {
      case CartAdd(‘id‘, items_) ⇒
        val newCart = CartCreated(id,items + items_)
        if (items disjointWith items_) newCart else error & newCart
      case CartRemove(‘id‘, items_) ⇒ CartCreated(id, items diff items_)
    }
  }
}
```

Property 4 formulated on XML messages directly

```
class Property4_XML extends Monitor[scala.xml.Elem] {
  Always {
    case add @ <CartAdd>{_*}</CartAdd> ⇒
      val c = getId(add)
      val items = getItems(add)
      for (i ∈ items) yield
        unless {
          case res @ <CartGetResponse>{_*}</CartGetResponse>
            if c == getId(res) ⇒ getItems(res) contains i
        } {
          case rem @ <CartRemove>{_*}</CartRemove>
            if c == getId(rem) &&
             (getItems(rem) contains i) ⇒ ok
        }
  }
}
```

Creating and applying a monitor

```
class Properties extends Monitor[Event] {
  monitor(
    new Property1(), new Property2(), new Property3(),
    new Property4(), new Property5())
}

object Main {
  def main(args: Array[String]) {
    val m = new Properties
    val file : String = "..."
    val xmlEvents = scala.xml.XML.loadFile( file )

    for (elem ∈ xmlEvents \ "_") {
      m.verify(xmlToObject(elem))
    }
    m.end()
  }
}
```

Implementation

Implementation

```
class Monitor[E <: AnyRef] {
  val monitorName = this.getClass().getSimpleName()

  var states : Set[state] = Set()

  var monitors : List[Monitor[E]] = List()

  def monitor(monitors:Monitor[E]*) {
    this.monitors ++= monitors
  }

   ...

}
```

Example: submonitors

```
class  Properties  extends Monitor[Event] {
  monitor(
    new Property1(), new Property2(), new Property3(),
    new Property4(), new Property5())
}

object Main {
  def main(args: Array[String]) {
    val m = new Properties
    val  file :  String  = "..."
    val xmlEvents = scala.xml.XML.loadFile( file )

    for  (elem ∈ xmlEvents \ "_") {
      m.verify(xmlToObject(elem))
    }
    m.end()
  }
}
```

Implementation

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

def noTransitions : Transitions = {
  case _ if false ⇒ null
}

val emptySet : Set[state] = Set()
```

Example: transitions and states

```
class Property5 extends Monitor[Event] {
  Always {
    case CartCreateResponse(c) ⇒ CartCreated(c)
    case CartAdd(c, _) if !CartCreated(c) ⇒ error
  }

  case class CartCreated(c: Int) extends state {
    Watch {
      case CartDelete(‘c‘) ⇒ ok
    }
  }
}
```

Implementation

```
class state {
  var transitions : Transitions = noTransitions
  var isFinal : Boolean = true

  def apply(event:E):Set[state] =
    if (transitions.isDefinedAt(event))
      transitions(event) else emptySet

  def Watch(ts: Transitions) {
    transitions = ts
  }

  def Always(ts: Transitions) {
    transitions = ts andThen (_ + this)
  }

  def Hot(ts: Transitions) {
   Watch(ts); isFinal = false
  }
```

Implementation

```
  def Wnext(ts: Transitions ) {
     transitions  = ts orElse {
        case _ ⇒ ok
     }
  }

  def Next(ts : Transitions ) {
    Wnext(ts); isFinal = false
  }

  def Unless(ts1 : Transitions )(ts2 : Transitions ) {
     transitions  = ts2 orElse
      (ts1 andThen (_ + this))
  }

  def Until(ts1 : Transitions )(ts2 : Transitions ) {
    Unless(ts1)(ts2 ); isFinal = false
  }
}
```

Implementation

```
case object ok extends state
case object error  extends state

def error(msg:String): state = {
   println("\n*** " + msg + "\n")
   error
}
```

Example: inlined states

```
class Property3 extends Monitor[Event] {
  Always {
    case CartCreate(items) ⇒ next {
      case CartCreateResponse(c) ⇒ always {
        case CartAdd(‘c‘, items_) ⇒ items disjointWith items_
      }
    }
  }
}
```

Implementation

```
def watch(ts: Transitions ) = new state {Watch(ts)}
def always(ts : Transitions ) = new state {Always(ts)}
def hot(ts : Transitions ) = new state {Hot(ts)}
def wnext(ts: Transitions ) = new state {Wnext(ts)}
def next( ts : Transitions ) = new state {Next(ts)}

def unless (ts1 : Transitions )( ts2 : Transitions ) =
  new state { Unless(ts1 )( ts2 ) }

def until (ts1 : Transitions )( ts2 : Transitions ) =
  new state { Until (ts1 )( ts2 ) }
```

Implementation

```
def  initial (s: state) { states += s }

def Always(ts: Transitions) {  initial (always(ts)) }

def Unless(ts1: Transitions)(ts2: Transitions) {
    initial (unless(ts1)(ts2))
}

 ...
```

Example: objects as Boolean predicates

```
class Property5 extends Monitor[Event] {
  Always {
    case CartCreateResponse(c) ⇒ CartCreated(c)
    case CartAdd(c, _) if !CartCreated(c) ⇒ error
  }

  case class CartCreated(c: Int ) extends state {
    Watch {
      case CartDelete(‘c‘) ⇒ ok
    }
  }
}
```

Implementation

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

Implementation

```
implicit def ss1(u:Unit):Set[state] = Set(ok)

implicit def ss2(b:Boolean):Set[state] = Set(if (b) ok else error)

implicit def ss3(s:state):Set[state] = Set(s)

implicit def ss4(ss:List[state]):Set[state] = ss.toSet

implicit def ss5(s1:state) = new {
  def &(s2:state):Set[state] = Set(s1, s2)
}

implicit def ss6(set:Set[state]) = new {
  def &(s:state):Set[state] = set + s
}
```

Implementation

```
def stateExists(p: PartialFunction[state, Boolean]): Boolean = {
  states exists (p orElse { case _ => false })
}
```

Implementation

```
var statesToRemove : Set[state] = Set()
var statesToAdd : Set[state] = Set()
```

Implementation

```
def verify(event:E) {
  for (sourceState ∈ states) {
    val targetStates = sourceState(event)
    if (!targetStates.isEmpty) {
      statesToRemove += sourceState
      for (targetState ∈ targetStates) {
        targetState match {
          case 'error' ⇒ println("*** " + monitorName + " error!")
          case 'ok' ⇒
          case _ ⇒ statesToAdd += targetState
        }
      }
    }
  }
  states −−= statesToRemove; states ++= statesToAdd
  statesToRemove = emptySet; statesToAdd = emptySet
  for (monitor ∈ monitors) {monitor.verify(event)}
}
```

Implementation

```
def end() {
  val hotStates = states filter (!_.isFinal)
  if (!hotStates.isEmpty) {
    println("hot " + monitorName + " states:")
    hotStates foreach println
  }
  for (monitor ∈ monitors) {
    monitor.end()
  }
}
```

Evaluation

Results

trace nr.	1	2	3	4	5	6	7
memory	1	1	5	30	100	500	5000
length	30,933	2,000,002	2,100,010	2,000,060	2,000,200	2,001,000	1,010,000
parsing	3 sec	45 sec	47 sec	46 sec	46 sec	46 sec	24 sec
LogFire	26 1:190	42 47:900	41 50:996	34 58:391	23 1:27:488	8 3:55:696	1 15:54:769
Rete/UL	38 816	109 18:428	75 28:141	41 48:524	14 2:26:983	4 8:25:867	0.4 43:33:366
Drools	10 3:97	8 4:1:758	9 3:47:535	9 3:34:648	8 4:14:497	7 4:36:608	3 5:4:505
Ruler	95 326	138 14:441	78 27:77	8 4:5:593	0.8 41:39:750	0.034 977:20:636	DNF
LogScope	17 1:842	15 2:11:908	7 4:54:605	2 21:42:389	0.4 76:17:341	0.09 369:25:312	0.01 2074:43:470
TrContract	48 645	69 28:851	37 57:428	6 5:58:497	0.9 36:29:594	0.036 919:5:134	DNF
Daut	49 631	84 23:847	86 24:338	89 22:432	90 22:298	86 23:287	80 12:612
Daut^{sos}	102 302	192 10:435	79 26:438	24 1:22:727	8 4:19:697	2 16:27:990	0.18 92:2:26
Daut^{int}	233 133	1715 1:166	770 2:729	373 5:368	195 10:236	54 36:929	5 3:6:560
Mop	595 52	1381 1:448	1559 347	1341 1:491	7143 280	7096 282	847 1:193

Conclusion

Conclusion

- We have seen the concept of data automata
- Implemented as an external as well as an internal DSL
- Internal DSL is simple but hard to optimize if shallow

Other challenges

Textual SysML modeling language

1. Should provide a textual alternative to graphic notation
2. Should support at least so-called parametric block diagrams
   - elements
   - relations
   - constraints
3. Should support constraint solving
4. Related work: Alloy, Formula (from MSR), Z
5. And one can now ask: why does this have to be a different world than the programming language mentioned above? We plan to experiment with internal Scala DSL.

Will programming and specification merge?

- Modern programming languages, such as Python, Scala, Fortress have many things in common with specification language such as VDM.

Will programming and specification merge?

- Modern programming languages, such as Python, Scala, Fortress have many things in common with specification language such as VDM.
- We see programming constructs such as:
  - functional programming combined with imperative programming
  - algebraic datatypes
  - sets, list and maps as built in data types with mathematic notation
  - predicate subtypes ($\mathbb{N} = \{i \in \mathbb{Z} \mid i \geqq 0\}$)
  - design by contract: pre/post conditions, invariants on state
  - session types
  - predicate logic, quantification over finite sets (as functions)

The six language elements

1. High-level programming constructs (like Scala, Python, ...)
2. Low-level programming constructs (like C, C++, ...)
3. Specification constructs (like VDM, Z, B, ...)
4. Support for verification, refinement
5. Support for definition of DSLs (internal as well as external)
6. Support for visualization (static as well as dynamic)

The verifying compiler for a new language

1. FM community designs new language
2. and its verifying compiler

The suggestion

1. Form a group of people, which can be joined by anyone
2. Open source programming language design/verifying compiler project
3. With project meetings etc.
4. A webpage for language design

The end