Modeling Seen as Programming

Klaus Havelund
NASA JPL, California Inst. of Technology, USA

System Design meets Equation-based Languages

September 21, 2012

Acknowledgements

Part of the work described in this publication was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

MSL

Landing


NASA
Jet Propulsion Laboratory
California Institute of Technology
Charles Bold
Fault Pro

3.5 million lines of C code

Terminology

Terminology

- model engineering

Terminology

- model engineering = engineering models

Terminology

- model engineering $=$ engineering models
- model-based engineering

Terminology

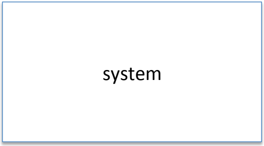
- model engineering = engineering models
- model-based engineering
- mode-based programming

Terminology

- model engineering = engineering models
- model-based engineering
- mode-based programming
- models, specifications used in software engineering (formal methods)

Runtime verification

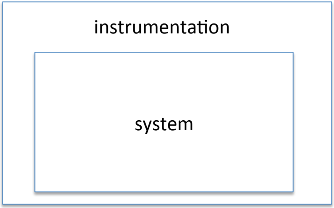
- Start with a system to monitor.



system

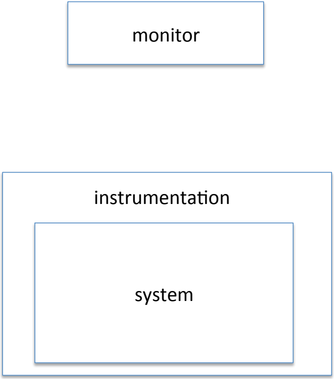
Runtime verification

- *Instrument* the system to record relevant events.



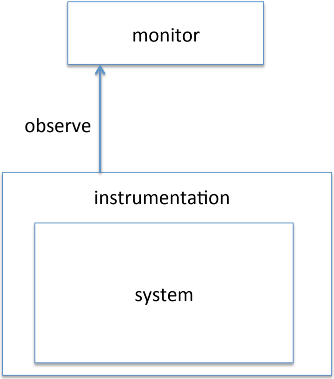
Runtime verification

- *Provide* a monitor.



Runtime verification

- *Dispatch* each received event to the monitor.


monitor

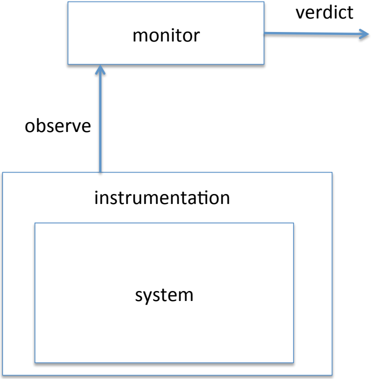
observe

instrumentation

system

Runtime verification

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



monitor

verdict

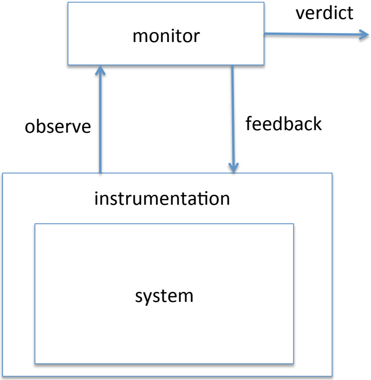
observe

instrumentation

system

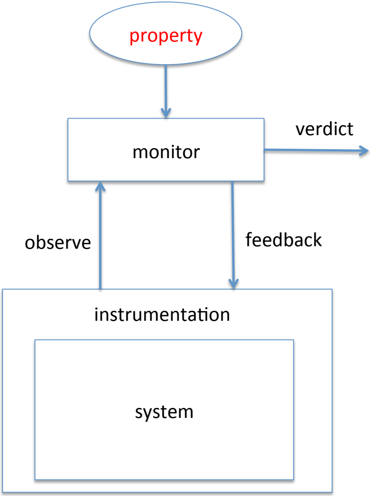
Runtime verification

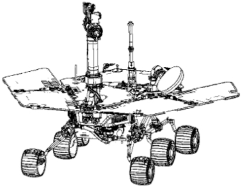
- Possibly generate *feedback* to the system.



Runtime verification

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





```
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)
```

requirements
relating events
across time

External versus internal DSL

External versus internal DSL

- External DSL

External versus internal DSL

- External DSL
  - small language typically with very focused functionality

External versus internal DSL

- External DSL
  - small language typically with very focused functionality
  - specialized parser

External versus internal DSL

- External DSL
  - small language typically with very focused functionality
  - specialized parser
  - pros:

External versus internal DSL

- External DSL
  - small language typically with very focused functionality
  - specialized parser
  - pros:
    - can be optimally succinct

External versus internal DSL

- External DSL
  - small language typically with very focused functionality
  - specialized parser
  - pros:
    - can be optimally succinct
    - "easy" to learn for person not familiar with programming language

External versus internal DSL

- External DSL
    - small language typically with very focused functionality
    - specialized parser
    - pros:
        - can be optimally succinct
        - “easy” to learn for person not familiar with programming language
        - analyzable: a spec can be analyzed easily, visualized, etc.

External versus internal DSL

- External DSL
  - small language typically with very focused functionality
  - specialized parser
  - pros:
    - ★ can be optimally succinct
    - ★ “easy” to learn for person not familiar with programming language
    - ★ analyzable: a spec can be analyzed easily, visualized, etc.
- Internal DSL

External versus internal DSL

- External DSL
  - small language typically with very focused functionality
  - specialized parser
  - pros:
    - can be optimally succinct
    - "easy" to learn for person not familiar with programming language
    - analyzable: a spec can be analyzed easily, visualized, etc.
- Internal DSL
  - an extension of an existing programming language

External versus internal DSL

- External DSL
  - small language typically with very focused functionality
  - specialized parser
  - pros:
    - can be optimally succinct
    - “easy” to learn for person not familiar with programming language
    - analyzable: a spec can be analyzed easily, visualized, etc.
- Internal DSL
  - an extension of an existing programming language
  - typically an API - using base language’s features only

External versus internal DSL

- External DSL
  - small language typically with very focused functionality
  - specialized parser
  - pros:
    - can be optimally succinct
    - “easy” to learn for person not familiar with programming language
    - analyzable: a spec can be analyzed easily, visualized, etc.
- Internal DSL
  - an extension of an existing programming language
  - typically an API - using base language’s features only
  - pros:

External versus internal DSL

- External DSL
  - small language typically with very focused functionality
  - specialized parser
  - pros:
    - can be optimally succinct
    - “easy” to learn for person not familiar with programming language
    - analyzable: a spec can be analyzed easily, visualized, etc.
- Internal DSL
  - an extension of an existing programming language
  - typically an API - using base language’s features only
  - pros:
    - easier to develop and later adapt

External versus internal DSL

- External DSL
  - small language typically with very focused functionality
  - specialized parser
  - pros:
    - can be optimally succinct
    - “easy” to learn for person not familiar with programming language
    - analyzable: a spec can be analyzed easily, visualized, etc.
- Internal DSL
  - an extension of an existing programming language
  - typically an API - using base language’s features only
  - pros:
    - easier to develop and later adapt
    - expressive, the programming language is never far away

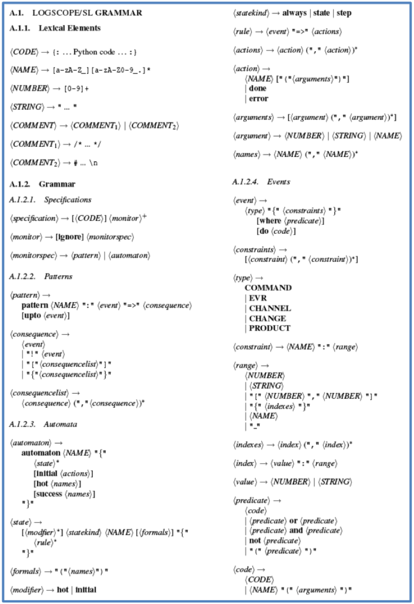
External versus internal DSL

- External DSL
  - small language typically with very focused functionality
  - specialized parser
  - pros:
    - can be optimally succinct
    - “easy” to learn for person not familiar with programming language
    - analyzable: a spec can be analyzed easily, visualized, etc.
- Internal DSL
  - an extension of an existing programming language
  - typically an API - using base language’s features only
  - pros:
    - easier to develop and later adapt
    - expressive, the programming language is never far away
    - allows use of existing tools such as type checkers, IDEs, etc.

External versus internal DSL

- External DSL: LogScope
  - small language typically with very focused functionality
  - specialized parser
  - pros:
    - can be optimally succinct
    - “easy” to learn for person not familiar with programming language
    - analyzable: a spec can be analyzed easily, visualized, etc.
- Internal DSL: TraceContract
  - an extension of an existing programming language
  - typically an API - using base language’s features only
  - pros:
    - easier to develop and later adapt
    - expressive, the programming language is never far away
    - allows use of existing tools such as type checkers, IDEs, etc.

LogScope V1 syntax



A.1. LOGSCOPE/SL GRAMMAR

A.1.1. Lexical Elements

```
⟨CODE⟩ → {: ... Python code ... :}

⟨NAME⟩ → [a-zA-Z_][a-zA-Z0-9_.]*

⟨NUMBER⟩ → [0-9]+

⟨STRING⟩ → " ... "

⟨COMMENT⟩ → ⟨COMMENT_1⟩ | ⟨COMMENT_2⟩

⟨COMMENT_1⟩ → /* ... */

⟨COMMENT_2⟩ → # ... \n
```

A.1.2. Grammar

A.1.2.1. Specifications

```
⟨specification⟩ → [⟨CODE⟩] ⟨monitor⟩+

⟨monitor⟩ → [ignore] ⟨monitorspec⟩

⟨monitorspec⟩ → ⟨pattern⟩ | ⟨automaton⟩
```

A.1.2.2. Patterns

```
⟨pattern⟩ →
    pattern ⟨NAME⟩ ":" ⟨event⟩ "=>" ⟨consequence⟩
    [upto ⟨event⟩]

⟨consequence⟩ →
    ⟨event⟩
    | "!" ⟨event⟩
    | "["⟨consequencelist⟩"]"
    | "{"⟨consequencelist⟩"}"

⟨consequencelist⟩ →
    ⟨consequence⟩ (","⟨consequence⟩)*
```

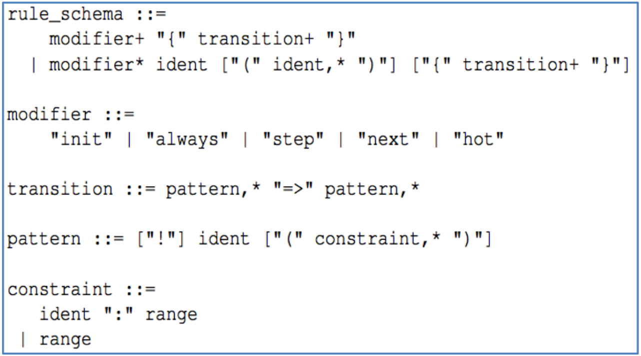
A.1.2.3. Automata

```
⟨automaton⟩ →
    automaton ⟨NAME⟩ "{"
        ⟨state⟩*
        [initial ⟨actions⟩]
        [hot ⟨names⟩]
        [success ⟨names⟩]
    "}"

⟨state⟩ →
    [⟨modfier⟩*] ⟨statekind⟩ ⟨NAME⟩ [⟨formals⟩] "{"
        ⟨rule⟩*
    "}"

⟨formals⟩ → "("⟨names⟩")"

⟨modifier⟩ → hot | initial

⟨statekind⟩ → always | state | step

⟨rule⟩ → ⟨event⟩ "=>" ⟨actions⟩

⟨actions⟩ → ⟨action⟩ ("," ⟨action⟩)*

⟨action⟩ →
    ⟨NAME⟩ ["("⟨arguments⟩")"]
    | done
    | error

⟨arguments⟩ → [⟨argument⟩ ("," ⟨argument⟩)*]

⟨argument⟩ → ⟨NUMBER⟩ | ⟨STRING⟩ | ⟨NAME⟩

⟨names⟩ → ⟨NAME⟩ ("," ⟨NAME⟩)*
```

A.1.2.4. Events

```
⟨event⟩ →
    ⟨type⟩ "{" ⟨constraints⟩ "}"
        [where ⟨predicate⟩]
        [do ⟨code⟩]

⟨constraints⟩ →
    [⟨constraint⟩ ("," ⟨constraint⟩)*]

⟨type⟩ →
    COMMAND
    | EVR
    | CHANNEL
    | CHANGE
    | PRODUCT

⟨constraint⟩ → ⟨NAME⟩ ":" ⟨range⟩

⟨range⟩ →
    ⟨NUMBER⟩
    | ⟨STRING⟩
    | "[" ⟨NUMBER⟩ "," ⟨NUMBER⟩ "]"
    | "{" ⟨indexes⟩ "}"
    | ⟨NAME⟩
    | "_"

⟨indexes⟩ → ⟨index⟩ ("," ⟨index⟩)*

⟨index⟩ → ⟨value⟩ ":" ⟨range⟩

⟨value⟩ → ⟨NUMBER⟩ | ⟨STRING⟩

⟨predicate⟩ →
    ⟨code⟩
    | ⟨predicate⟩ or ⟨predicate⟩
    | ⟨predicate⟩ and ⟨predicate⟩
    | not ⟨predicate⟩
    | "(" ⟨predicate⟩ ")"

⟨code⟩ →
    ⟨CODE⟩
    | ⟨NAME⟩ "(" ⟨arguments⟩ ")"
```

LogScope V2 syntax



```
rule_schema ::=
    modifier+ "{" transition+ "}"
  | modifier* ident ["(" ident,* ")"] ["{" transition+ "}"]

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

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

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

constraint ::=
   ident ":" range
 | range
```

Quote

Hemmingway & Hotchner, 1920ies:

If you are lucky enough to have lived in Paris as a young man, then wherever you go for the rest of your life, it stays with you, for Paris is a moveable feast.

Quote

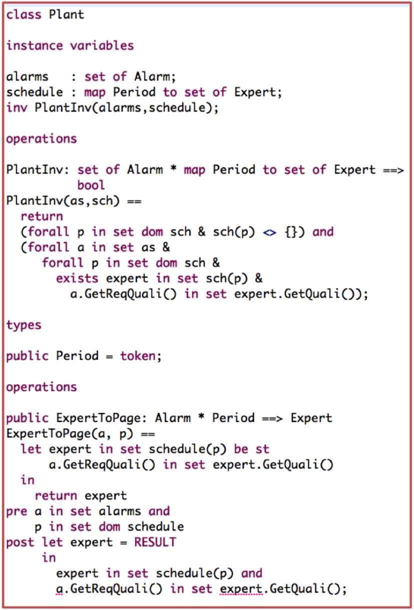
Havelund, 2012:

If you are lucky enough to have explored VDM as a young man, then wherever you go for the rest of your life, it stays with you, for VDM is a moveable feast.

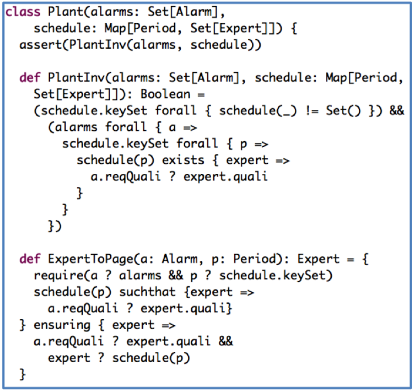
What is VDM?

- Combination of imperative and functional programming (data types, pattern matching, curried functions, lambda abstractions, side effects, loops, exceptions,)
- Design-by-contract: pre/post conditions + invariants
- Predicate subtypes
- Non-deterministic expressions (let x be such that P(x))
- First order predicate logic as Boolean expressions: universal and existential quantification
- Sets, lists and maps as built-in data types
- VDM^{++} added object orientation (Nico Plat et. al)

Chemical plant model in VDM versus Scala



```
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();
```



```
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
          }
        }
      })

  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)
  }
```

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

Commands must succeed

- We are analyzing log files containing information about commands being issued, and their success and failure respectively.

Requirement CommandMustSucceed

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

Property in LogScope

- For comparison we first show spec in the external DSL: LOGSCOPE.
- a **hot** state must be exited before end of log (non-final state).

```
automaton CommandMustSucceed {
  always {
    Command(n,x) => RequireSuccess(n,x)
  }

  hot RequireSuccess(name,number) {
    Fail (name,number) => error
    Success(name,number) => ok
  }
}
```

Property in LogScope

- Using LogScope's temporal logic layer.

```
pattern CommandMustSucceed:
  Command(n,x) =>
    [
      !  Fail (n,x),
         Success(n,x),
    ]
```

Events in TraceContract

- First we need to define the events we observe:
    - commands being issued, each having a name and a number
    - successes of commands
    - failures of commands
- Each event type sub-classes a type: Event
- **case**-classes allow for pattern matching over objects of the class

```
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
```

Property in TraceContract - looks very similar

- Uses partial functions: {**case** ... => ...} defined with pattern matching as arguments to DSL functions (*require* and *hot*) defined in *Monitor* class. *RequireSuccess* is a user-defined function representing a state.
- A quoted name, such as ‘name‘ represents the *value of* that name.

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

  def RequireSuccess(name: String, number: Int) =
    hot {
      case Fail(‘name‘, ‘number‘) => error
      case Success(‘name‘, ‘number‘) => ok
    }
}
```

Property in TraceContract - looks very similar

- Uses partial functions: {**case** ... =>...} defined with pattern matching as arguments to DSL functions (*require* and *hot*) defined in *Monitor* class. *RequireSuccess* is a user-defined function representing a state.
- A quoted name, such as ‘name‘ represents the *value of* that name.

```
class CommandMustSucceed extends Monitor[Event] {
  require {
    case Command(n, x) => RequireSuccess(n, x)
  }

  def RequireSuccess(name: String, number: Int) =
    hot {
      case Fail (‘name‘, ‘number‘) => error
      case Success(‘name‘, ‘number‘) => ok
    }
}
```

Inlining the call of *RequireSuccess(n,x)*

- Since *RequireSuccess(n, x)* is a function, the call of it can be inlined.
- After all, this is “just” a program and standard program transformation works.
- The result is an interesting temporal logic like specification with an un-named hot state.

```
class  CommandMustSucceed extends Monitor[Event] {
  require  {
    case Command(n, x) =>
      hot {
        case  Fail (‘n‘,  ‘x‘)  => error
        case Success(‘n‘,  ‘x‘)  => ok
      }
  }
}
```

Same property in LTL

- TRACECONTRACT also offers future time linear temporal logic (LTL).
- allowing to write events as formulas, negations, propositional formulas, and temporal.
- ϕ until ψ means: ψ must eventually hold, and until then ϕ must hold.

```
class CommandMustSucceed extends Monitor[Event] {
  require {
    case Command(n, x) =>
      not(Fail(n, x)) until (Success(n, x))
  }
}
```

Same property in LTL

- TRACECONTRACT also offers future time linear temporal logic (LTL).
- allowing to write events as formulas, negations, propositional formulas, and temporal.
- ϕ until ψ means: ψ must eventually hold, and until then ϕ must hold.

```
class CommandMustSucceed extends Monitor[Event] {
  require {
    case Command(n, x) =>
      not(Fail(n, x)) until (Success(n, x))
  }
}
```

- note mix of `Scala`'s pattern matching (to catch arguments of command) and LTL.

Success of power commands

Requirement PowerCommandSuccess

Power commands must succeed within 10 seconds.

Property in LogScope

- Defining and using Python predicates in LogScope.

```
{:
def within(t1,t2,max):
  return (t2-t1) <= max
:}

pattern PowerCommands:
  Command(n, x, t1) where {: n.startswith(”PWR”) :} =>
    Success(n, x, t2) where {: within(t1,t2,10000) :}
```

Same property in TraceContract

- TRACECONTRACT allows direct integration of code and formulas.

```
class PowerCommands extends Monitor[Event] {
  def within(t1: Int,t2: Int, max: Int) = (t2−t1) <= max

   require {
     case Command(n, x, t1) if n.startsWith(”PWR”) =>
       hot {
         case Success(‘n‘, ‘x‘, t2) if within(t1,t2,10000) => ok
       }
   }
}
```

10 first commands must succeed

Requirement First10CommandsMustSucceed

The first 10 issued commands must succeed, without a failure to occur before then.

Counting: first 10 commands must succeed

- Code (here counting and testing on counter) can be mixed with logic.
- That is: increase counter and return LTL formula.

```
class First10CommandsMustSucceed extends Monitor[Event] {
  var count = 0
  require {
    case Command(n, x) if count < 10 =>
      count = count + 1
      not( Fail (n, x)) until (Success(n, x))
  }
}
```

long sequence

Requirement CommandSequence

Whenever a flight software command is issued, there should follow a dispatch and then exactly one success.
No dispatch failure before the dispatch, and
no failure between dispatch and success.

Property in LogScope

- Using LOGSCOPE's sequence operator.

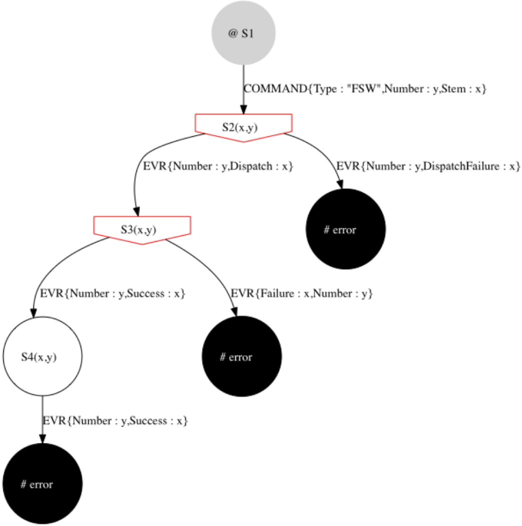
```
pattern CommandSequence:
  Command(n,x) =>
    [
      ! DispatchFailure(n,x),
        Dispatch(n,x),
      ! Fail(n,x),
        Success(n,x),
      ! Success(n,x)
    ]
```

Same property in TraceContract

- TraceContract allows mixing of states.

```
class CommandSequence extends Monitor[Event] {
  require {
    case Command(n, x) =>
      hot {
        case DispatchFailure(‘n‘, ‘x‘) => error
        case Dispatch(‘n‘, ‘x‘) =>
          hot {
            case Fail(‘n‘, ‘x‘) => error
            case Success(‘n‘, ‘x‘) =>
              state {
                case Success(‘n‘, ‘x‘) => error
              }
          }
      }
  }
```

Visualization of LogScope statemachine



Much more difficult to do with internal DSL such as TraceContract.

Property that we cannot write in LogScope

- Antecedent (condition) containing multiple events.

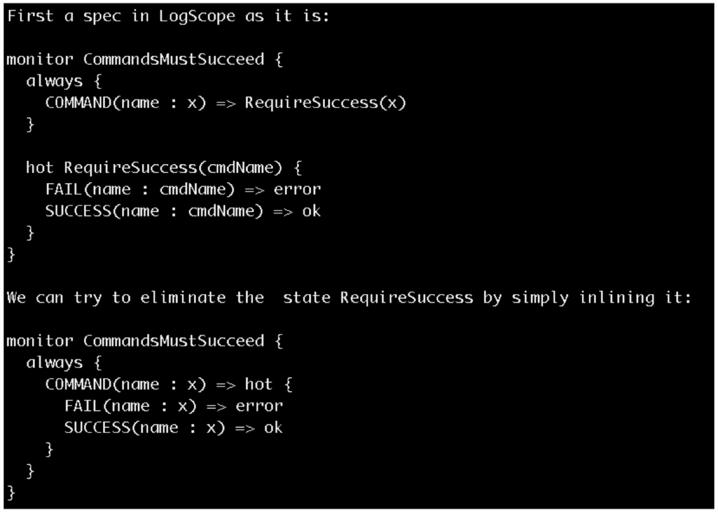
```
pattern CommandSequenceAsCondition:
    [
        Command(n,x),
      ! DispatchFailure(n,x),
        Dispatch(n,x)
    ]
    =>
    [
      ! Fail(n,x),
        Success(n,x),
      ! Success(n,x)
    ]
```

However we can write it in TraceContract

- TRACECONTRACT by just changing one of the state modifiers.

```
class CommandSequence extends Monitor[Event] {
  require {
    case Command(n, x) =>
      state {
        case DispatchFailure(`n`, `x`) => error
        case Dispatch(`n`, `x`) =>
          hot {
            case Fail(`n`, `x`) => error
            case Success(`n`, `x`) =>
              state {
                case Success(`n`, `x`) => error
              }
          }
      }
  }
```

Some notes from a notebook - before TraceContract



```
First a spec in LogScope as it is:

monitor CommandsMustSucceed {
  always {
    COMMAND(name : x) => RequireSuccess(x)
  }

  hot RequireSuccess(cmdName) {
    FAIL(name : cmdName) => error
    SUCCESS(name : cmdName) => ok
  }
}

We can try to eliminate the  state RequireSuccess by simply inlining it:

monitor CommandsMustSucceed {
  always {
    COMMAND(name : x) => hot {
      FAIL(name : x) => error
      SUCCESS(name : x) => ok
    }
  }
}
```

TraceContract later offered this feature.

Alternation

Requirement AlternatingCommandSuccess

Commands and successes should alternate.

State machine solution

```
class AlternatingCommandSuccess extends Monitor[Event] {
  property(s1)

  def s1: Formula =
    state {
      case Command(n, x) => s2(n, x)
      case _ => error
    }

  def s2(name: String, number: Int) =
    state {
      case Success(‘name‘, ‘number‘) => s1
      case _ => error
    }
}
```

State machine solution - with next-states

```
class AlternatingCommandSuccess extends Monitor[Event] {
  property(s1)

  def s1: Formula =
    next {
      case Command(n, x) => s2(n, x)
    }

  def s2(name: String, number: Int) =
    next {
      case Success(`name`, `number`) => s1
    }
}
```

A past time property

- Properties so far have been future time properties: from some event, the future behavior must satisfy some property.
- The following requirement refers to the past of some event (success).

Requirement SuccessHasAReason

A success must be caused by a previously issued command.

TraceContract offers limited rule-based programming

- State logic and LTL cannot express this property.

TraceContract offers limited rule-based programming

- State logic and LTL cannot express this property.
- TRACECONTRACT offers a limited form of rule-based programming, were a fact f (sub-classing class *Fact*) can be queried ($f?$), created ($f+$), and deleted ($f-$). The result in the latter two cases is True.

TraceContract offers limited rule-based programming

- State logic and LTL cannot express this property.
- TRACECONTRACT offers a limited form of rule-based programming, were a fact *f* (sub-classing class *Fact*) can be queried (*f*?), created (*f*+), and deleted (*f*−). The result in the latter two cases is True.

```
class SuccessHasAReason extends Monitor[Event] {
  case class Commanded(name: String, nr: Int) extends Fact

   require {
     case Command(n, x) => Commanded(n, x) +
     case Success(n, x) =>
       if (Commanded(n, x) ?)
         Commanded(n, x) −
       else
         error
   }
}
```

The ?- abbreviation

- We can we make this monitor simpler by using test-and-set: f ?−, for a given fact f, meaning: *return true iff. the fact f is recorded, delete the fact in any case*.

```
class  SuccessHasAReason extends Monitor[Event] {
  case class  Commanded(name: String, nr: Int) extends Fact

   require  {
     case Command(n, x) => Commanded(n, x) +
     case Success(n,  x)  => Commanded(n, x) ?−
   }
}
```

Making monitors of monitors

- We can create a new monitor which includes other monitors as sub-monitors. Useful for organizing properties.
- The semantics is the obvious one of conjunction: all monitors will get checked individually.

```
class CommandRequirements extends Monitor[Event] {
  monitor(
    new CommandMustSucceed,
    new MaxOneSuccess,
    new SuccessHasAReason)
}
```

Analyzing a complete trace (log analysis)

- To verify a trace: first create it, then instantiate monitor, and call *verify* method on monitor with trace as argument.

```
object TraceAnalysis extends Application {
  val trace: List[Event] =
    List(
      Command("STOP_DRIVING", 1),
      Command("TAKE_PICTURE", 2),
      Fail("STOP_DRIVING", 1),
      Success("TAKE_PICTURE", 2),
      Success("SEND_TELEMETRY", 42))

  val monitor = new CommandRequirements
  monitor.verify(trace)
}
```

Alternatively: analyzing event by event (online monitoring)

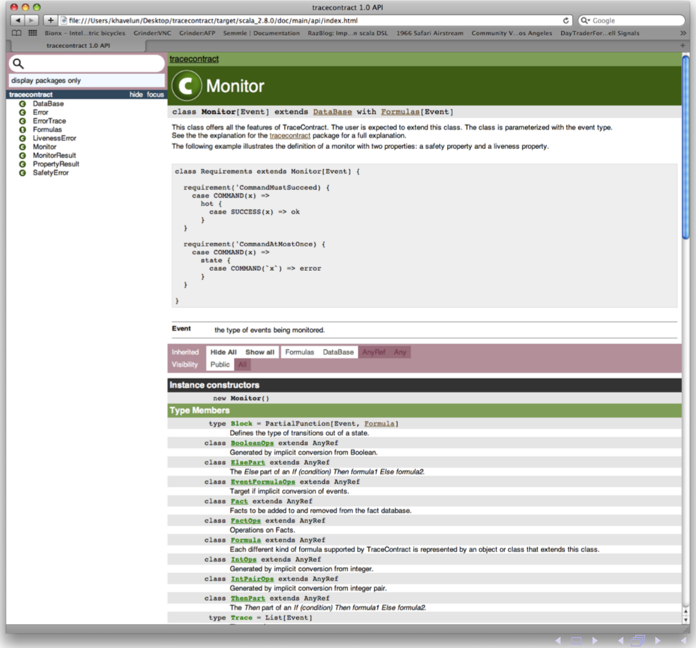
- To verify a sequence of events: instantiate monitor, and call *verify* method on monitor for each event, and call *end()* if event flow terminates.

```
object TraceAnalysis extends Application {
  val monitor = new CommandRequirements
  monitor. verify (Command("STOP_DRIVING", 1))
  monitor. verify (Command("TAKE_PICTURE", 2))
  monitor. verify ( Fail ("STOP_DRIVING", 1))
  monitor. verify (Success("TAKE_PICTURE", 2))
  monitor. verify (Success("SEND_TELEMETRY", 42))
  monitor.end()
}
```

Result

```
CommandMustSucceed property violated
Violating event number 3: Fail(STOP_DRIVING,1)
Error trace:
  1=Command(STOP_DRIVING,1)
  3=Fail(STOP_DRIVING,1)


SuccessHasAReason property violated
Violating event number 5: Success(SEND_TELEMETRY,42)
Error trace:
  5=Success(SEND_TELEMETRY,42)
```

ScalaDoc documentation of API



tracecontract

- DataBase
- Error
- ErrorTrace
- Formulas
- LivenessError
- Monitor
- MonitorResult
- PropertyResult
- SafetyError

tracecontract

Monitor

`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.

```
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.

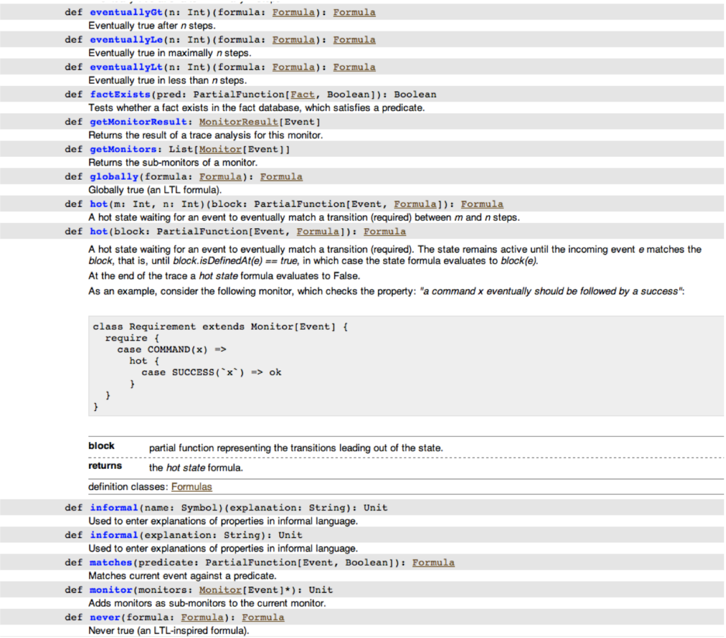
```
class IntOps extends AnyRef
```
Generated by implicit conversion from integer.

```
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]
```

ScalaDoc documentation of API



`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). 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"*:

```
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.

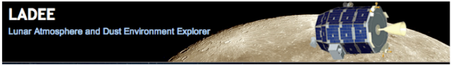
`def monitor(monitors: Monitor[Event]*): Unit`

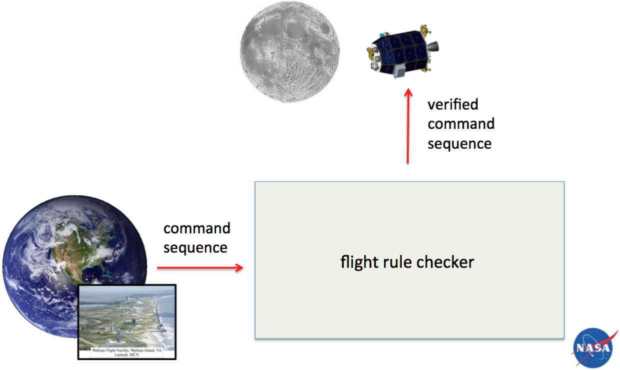
Adds monitors as sub-monitors to the current monitor.

`def never(formula: Formula): Formula`

Never true (an LTL-inspired formula).

LADEE mission



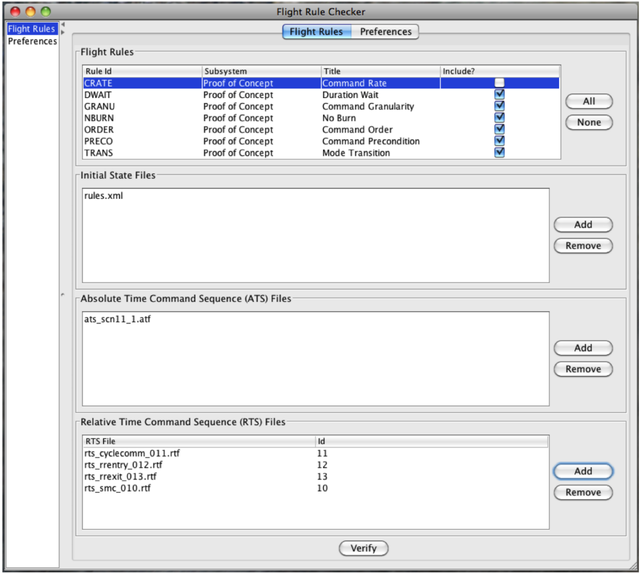


verified command sequence

command sequence

flight rule checker

GUI interface to TraceContract (LADEE mission)



Flight Rules

Rule Id	Subsystem	Title	Include?
CRATE	Proof of Concept	Command Rate	☐
DWAIT	Proof of Concept	Duration Wait	☑
GRANU	Proof of Concept	Command Granularity	☑
NBURN	Proof of Concept	No Burn	☑
ORDER	Proof of Concept	Command Order	☑
PRECO	Proof of Concept	Command Precondition	☑
TRANS	Proof of Concept	Mode Transition	☑

Initial State Files

rules.xml

Absolute Time Command Sequence (ATS) Files

ats_scn11_1.atf

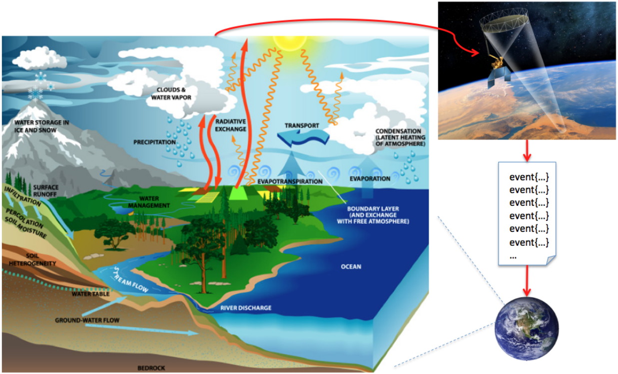
Relative Time Command Sequence (RTS) Files

RTS File	Id
rts_cyclecomm_011.rtf	11
rts_rrentry_012.rtf	12
rts_rrexit_013.rtf	13
rts_smc_010.rtf	10

SMAP mission

SMAP

mapping of soil moisture and its freeze/thaw state



event{...}
event{...}
event{...}
event{...}
event{...}
event{...}
...

Definition of parameterized monitors

```
class CommandSuccess(cmd: String, success: Boolean = true)
extends Monitor[Event] {
     require {
       case Command(‘cmd‘,number) =>
         hot {
           case Success(‘cmd‘,‘number‘)  => success
           case Fail (‘cmd‘,‘number‘)  => !success
         }
     }
}

monitor(new CommandSuccess(”STOP”))
```

Summary

- TRACECONTRACT is an API.
- Very expressive and convenient for programmers to use.
- For this reason mainly it has been adopted by practitioners.
- Has very simple implementation, which is easy to modify.
- Change requests are easy to process.
- It is, however, difficult to analyze a TRACECONTRACT specification since it fundamentally is a `Scala` program - requires some form of reflection or interaction with compiler.
- It will not be suitable for non-`Scala` programmers.