From Specifications to Monitors

Klaus Havelund (NASA Jet Propulsion Laboratory/Caltech, USA)
Doron Peled (Bar Ilan University, Israel)
Dogan Ulus (Verimag/Universite Grenoble-Alpes, France)

Workshop on Software Correctness and Reliability
October 13-14, 2017
ETH, Zurich, Switzerland
Definition (Runtime Verification)
Runtime Verification is the discipline of computer science dedicated to the analysis of system executions, including checking them against formalized specifications.

Alternative formulation: “get as much out of your runs as possible”:
- verification of execution traces, Boolean true or false
- collection of statistics, beyond the Boolean domain
- specification learning
- analysis with algorithms (no specs): data race and deadlock analysis
- trace visualization
- fault protection: changing behavior
Runtime verification

\[ M : \mathcal{E}^* \rightarrow D \]

\[ M : \mathcal{P}(\mathcal{E}^*) \rightarrow D \]
fault protection

response

monitor
log file analysis
command sequence analysis

monitor
command sequence analysis
In April 2011 TraceContract was selected by LADEE mission management for writing the flight rule checker!
Classical dimensions to consider = $E^3$

- Efficiency
- Expressiveness
- Elegance
DejaVu

With:
Doron Peled (Bar Ilan University, Israel)
Dogan Ulus (Verimag/Universite Grenoble-Alpes, France)
First-order past time temporal formulas

$$\forall f \ (\text{close}(f) \rightarrow \text{Popen}(f))$$

$$\forall f \ (\text{close}(f) \rightarrow \oplus(\neg \text{close}(f) \ S \text{open}(f)))$$
The Logic

\[ \varphi ::= \text{true} \mid p(t_1, \ldots, t_n) \mid \neg \varphi \mid \varphi_1 \lor \varphi_2 \mid \exists x \bullet \varphi \mid \Theta \varphi \mid \varphi_1 \mathcal{S} \varphi_2 \]

\[ t ::= c \mid x \]

Derived Constructs

- \( \text{false} = \neg \text{true} \)
- \( \varphi_1 \land \varphi_2 = \neg (\neg \varphi_1 \lor \neg \varphi_2) \)
- \( \varphi_1 \Rightarrow \varphi_2 = \neg \varphi_1 \lor \varphi_2 \)
- \( \forall x \bullet \varphi = \neg \exists x \bullet \neg \varphi \)
- \( \mathcal{P} \varphi = \text{true} \mathcal{S} \varphi \)
- \( \mathcal{H} \varphi = \neg \mathcal{P} \neg \varphi \)
- \( [\varphi_1, \varphi_2] = (\neg \varphi_2) \mathcal{S} \varphi_1 \)

Example

\[ \forall \text{user} \bullet \forall \text{file} \bullet \]
\[ \text{access(user, file)} \Rightarrow [\text{login(user)}, \text{logout(user)}] \]
\[ \land [\text{open(file)}, \text{close(file)}] \]
Result of verifying trace against a formula

true or false

a set of assignments

a BDD
Some definitions

• **Domains** \( D_1, D_2, \ldots, \) possibly infinite

• **Variables** \( V = \{x, y, \ldots\} \) ranging over domains \( x : D_1, y : D_2, \ldots \)

• **Assignments** \([x \to \text{“tel”}, y \to \text{“tel2”}]\)

• **Predicates** \( \text{open(“tel”), open(x), close(y), …} \)

• **Ground predicates** \( \text{open(“tel”)}\)

• **A state** is a set of ground predicates: \( \{\text{open(“tel1”), open(“tel2”)}\} \)

• **A trace** is a finite sequence of states: \( <s_1, s_2, \ldots, s_n> \)
First Semantics: the “standard” definition

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

\[ \forall f \ (\text{close}(f) \rightarrow \text{Popen}(f)) \]

We need to save all past values of file names that were opened, and compare with the current one that is closed.
Let's look at a more complicated formula:
\[ \exists x \exists y \ (q(y) \land p(x)) \]

The answer is \( F \): there is no common value of \( q(y) \) since \( q(5) \).
Let's look at a more complicated formula:

$$\exists x \exists y \ (q(y) \land p(x))$$

The answer is \textbf{T}: there is a common value of $q(9)$ since $p(3)$. 
\[ \exists x \exists y \ (q(y) \land p(x)) \]
The “bookkeeping” is nontrivial:

The answer is \( T \): there is a common value of \( q(9) \) since \( p(3) \).

Keep common subsets of values of \( y \) in \( q(y) \) since you see \( p(5) \).
Keep common subsets of values of \( y \) in \( q(y) \) since you see \( p(3) \).
\[ \exists x \, \exists y \, (q(y) \land p(x)) \]

The “bookkeeping” is nontrivial:

The answer is \( T \): there is a common value of \( q(9) \) since \( p(3) \).

Keep common subsets of values of \( y \) in \( q(y) \) since you see \( p(3) \).

Do the same with \( x=5 \)

In general we keep track of sets of tuples (assignments) of \( x \) and \( y \) values: e.g. \{\( (3,7), (3,8), (3,9) \}\}, at each point.

Standard semantics does not give a good intuition how to perform this bookkeeping!
Second semantics: Set semantics. Each (sub)formula on a prefix of an execution denotes a set of assignments that satisfy the formula.

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

Theorem:

$\gamma \in I[\varphi, \sigma, i]$ iff $(\gamma, \sigma, i) \models \varphi$. 
Third Semantics:
representing sets of assignments as BDDs

\[ \neg(p) \land q \land \neg(r) \]

\[ (p \land q \land r) \lor (p \land q \land \neg(r)) \]
We do not represent values directly. Instead we **enumerate** values in binary, and store sets of these binaries as BDDs.

\[ \forall f \ (\text{close}(f) \rightarrow \text{P open}(f)) \]

We keep values in a **hash** to check reoccurrence.
Value to bit string table:

- "tel" → 000
- "dict" → 001
- "out" → 010
- "tel2" → 011

Diagram:

- 000: b0, b1, b2
- 001: b0, b1, b2
- 010: b0, b1, b2
- 011: b0, b1, b2

Each node corresponds to a bit position in the string.
Characteristic function for our bit vector set representing the accumulated set of values $P \text{ open}(f)$

\{“tel”, “dict”, “out”\}

\{000, 001, 010\}

\{000\} union \{001\} union \{010\}

$BDD(000) \text{ or } BDD(001) \text{ or } BDD(010)$

But not $BDD(011)$ (for “tel2”)

Numerations $\geq 100$ are for values not seen so far.

$$\lambda(b_0 b_1 b_2).$$

$$(!b_0 \land !b_1 \land !b_2) \lor (!b_0 \land !b_1 \land b_2) \lor (!b_0 \land b_1 \land !b_2)$$
Characteristic function for our bit vector set

We account for values not seen so far.

As long as we use $n$ bits and there will be less than $2^n$ values, then the higher enumerations represent values not seen so far.

In particular, the value $11\ldots111$ represents “all values not yet seen”.

We can negate, obtaining the BDD for $\neg P \text{ open}(f)$

*This is easy: replace $F$ by $T$ at leaf level.*

We can start with a rather large value of $n$, hoping that the BDD will be compact.

We may also add a bit “on the fly”, when more than $2^n$ values occur.
Representing a set of assignments using enumerations

\[
\{ [x \rightarrow \text{“a”}, y \rightarrow 42], \\
[x \rightarrow \text{“b”}, y \rightarrow 52] \}
\]

\[
\begin{align*}
\text{forall } x . \text{forall } y . \\
\text{send}(x,y) & \rightarrow P \text{recv}(x,y) \\
\text{recv(“a”),42} & \\
\text{recv(“b”),52} \\
\ldots
\end{align*}
\]

\[
\begin{array}{c}
x \rightarrow \\
\text{“a”} & \rightarrow 000 \\
\text{“b”} & \rightarrow 001
\end{array}
\]

\[
\begin{array}{c}
y \rightarrow \\
42 & \rightarrow 000 \\
52 & \rightarrow 001
\end{array}
\]
Representing a set of assignments

\[
\{ \\
[x \rightarrow \text{"a"}, y \rightarrow 42], \\
[x \rightarrow \text{"b"}, y \rightarrow 52] \\
\}
\]

\[
\begin{align*}
x & \rightarrow \text{"a"} & \rightarrow 000 \\
& \rightarrow \text{"b"} & \rightarrow 001 \\
y & \rightarrow 42 & \rightarrow 000 \\
& \rightarrow 52 & \rightarrow 001
\end{align*}
\]

\[
\begin{align*}
x_0x_1x_2y_0y_1y_2 & \rightarrow 0000000 \\
& \text{or} \\
& \rightarrow 001001
\end{align*}
\]
Representing a set of assignments

\[
\{ \\
[ x \rightarrow \text{“a”} \ , \ y \rightarrow 42] \ , \\
[ x \rightarrow \text{“b”} \ , \ y \rightarrow 52] \\
\}
\]

\[
x_0 x_1 x_2 y_0 y_1 y_2 \\
0 0 0 0 0 0 \\
\text{or} \\
0 0 1 0 0 1 \\
\]

\[
x \rightarrow \\
\begin{cases} 
\text{“a”} \rightarrow 000 \\
\text{“b”} \rightarrow 001 
\end{cases}
\]

\[
y \rightarrow \\
\begin{cases} 
42 \rightarrow 000 \\
52 \rightarrow 001 
\end{cases}
\]
Representing a set of assignments

\[
\{ \\
[ x \to "a", y \to 42] , \\
[ x \to "b", y \to 52] 
\}
\]

\[
\begin{align*}
\text{x} & \rightarrow \begin{cases} 
"a" & \rightarrow 000 \\
"b" & \rightarrow 001 
\end{cases} \\
\text{y} & \rightarrow \begin{cases} 
42 & \rightarrow 000 \\
52 & \rightarrow 001 
\end{cases}
\end{align*}
\]

\[
\begin{align*}
x_0 x_1 x_2 y_0 y_1 y_2 & \\
\text{or} & \\
0 0 0 0 0 0 \\
0 0 1 0 0 1
\end{align*}
\]
Looking back at the set semantics: since every assignment is a BDD

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

We can replace the set operations with BDD operations:
Union \( \cup \) by disjunction \( \lor \), Intersection \( \cap \) by conjunction \( \land \), hide is existential quantification over all bits of variable.
Algorithm

1) Initially, for each subformula $\varphi$, $\text{now}(\varphi) = \text{BDD}(0)$.
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 $\text{now}(\psi)$.
   - $\text{now}({\text{true}}) = \text{BDD}(1)$
   - $\text{now}({\text{p(a)}}) = \text{if } p(a) \in s \text{ then } \text{BDD}(1) \text{ else } \text{BDD}(0)$
   - $\text{now}({\text{\exists p(a)}}) = \text{if } \exists a p(a) \in s \text{ then } \text{build}(x, a) \text{ else } \text{BDD}(0)$
   - $\text{now}((\varphi \land \psi)) = \text{and}(\text{now}(\varphi), \text{now}(\psi))$
   - $\text{now}(\neg \varphi) = \text{not}(\text{now}(\varphi))$
   - $\text{now}((\varphi S \psi)) = \text{or}(\text{now}(\psi), \text{and}(\text{now}(\varphi), \text{pre}((\varphi S \psi))))$
   - $\text{now}(\ominus \varphi) = \text{pre}(\varphi)$
   - $\text{now}(\exists x \varphi) = \text{exists}(\langle x_0, \ldots, x_{k-1} \rangle, \text{now}(\varphi))$

5) Goto step 2.
Limiting quantification

• Limiting quantification to seen values:

\[ \exists x \neg P \ g(x) \]

\[ \exists x \ (\text{seen}(x) \land \neg P \ g(x)) \]

• Finite domains:

\[ \text{smaller}(y, 3) = \neg(y_0 \land y_1) \]

\[ \exists x \ (\text{smaller}(x, m) \land \varphi) \]

\[ \forall x \ (\text{smaller}(x, m) \rightarrow \varphi) \]
Implementation: DejaVu
JavaBDD

JavaBDD is a Java library for manipulating BDDs (Binary Decision Diagrams). Binary decision diagrams are widely used in model checking, formal verification, optimizing circuit diagrams, etc. For an excellent overview of the BDD data structure, see this set of lecture notes by Henrik Reif Andersen.

The JavaBDD API is based on that of the popular BuDDy package, a BDD package written in C by J?rn Lind-Nielsen. However, JavaBDD's API is designed to be object-oriented. The ugly C function interface and reference counting schemes have been hidden underneath a uniform, object-oriented interface.

JavaBDD includes a 100% Java implementation. It can also interface with the JDD library, or with three popular BDD libraries written in C via a JNI interface: BuDDy, CUDD, and CAL. JavaBDD provides a uniform interface to all of these libraries, so you can easily switch between them without having to make changes to your application.

JavaBDD is designed for high performance applications, so it also exposes many of the lower level options of the BDD library, like cache sizes and advanced variable reordering.
**Architecture**

```
prop secure :
  forall (user) forall (file)
  access(user, file) ->
    [login(user), logout(user))
    &
    [open(file), close(file))
```

Scala parser combinators

```
login, John
open, tel
access, John, tel
close, tel
access, John, tel
logout, John
```

Apache commons CSV
(Comma Separated Value format) parser

```
*** Property secure violated on event number 5:
  access(John, tel)
```
dejavu <specFile> <logFile> [<bitsPerVariable>]

Grammar:

```plaintext
<spec>  ::=  <prop> . . .  <prop>
<prop>  ::=  'prop'  <id>  ':'  <form>
<form>  ::=  'true'  |  'false'
           |  <id>  [  '('  <param>  ','  ...  ','  <param>  ')'  ]
           |  <form>  <binop>  <form>
           |  '['  <form>  ','  <form>  ']'  
           |  <unop>  <form>
           |  ('exists'  |  'forall')  <id>  '.'  <form>
           |  '('  <form>  ')
<binop>  ::=  '-'  |  '|'  |  '&'  |  'S'
<unop>  ::=  '!'  |  '@'  |  'P'  |  'H'
<param>  ::=  <id>  |  <string>  |  <integer>
```
prop p: forall f . close(f) → exists m . P 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
    }
}
prop p: \(\forall f. \text{close}(f) \rightarrow \exists m. \text{P open}(f,m)\)
prop p: \( \forall f \cdot \text{close}(f) \rightarrow \exists m \cdot \text{P open}(f,m) \)
prop p: forall f . close(f) \rightarrow exists m . P open(f,m)
open, input, read, open, output, write, close, out

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

close, out

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

\[ \text{prop } p : \forall f \cdot \text{close}(f) \rightarrow \exists m \cdot P \text{ open}(f, m) \]
prop p: forall f . close(f) → exists m . P open(f,m)
prop \( p \): \( \forall f \) . \( \text{close}(f) \rightarrow \exists m . \text{P open}(f,m) \)
prop p: \forall f . \text{close}(f) \rightarrow \exists m . \text{P open}(f,m)
\[
\text{prop } p: \text{forall } f . \ close(f) \rightarrow \text{exists } m . \ P \ open(f,m)
\]
prop: forall f . close(f) \rightarrow exists m . P open(f,m)
Evaluation Properties in QTL

\textbf{prop} access : \textbf{forall} u . \textbf{forall} f .
\hspace{1em} access(u, f) \rightarrow [\text{login}(u), \text{logout}(u)] \& [\text{open}(f), \text{close}(f)]

\textbf{prop} file : \textbf{forall} f .
\hspace{1em} close(f) \rightarrow \textbf{exists} m . @ [\text{open}(f, m), \text{close}(f)]

\textbf{prop} fifo : \textbf{forall} x .
\hspace{1em} (\text{enter}(x) \rightarrow \neg \exists P \text{ enter}(x)) \&
\hspace{1em} (\text{exit}(x) \rightarrow \neg \exists P \text{ exit}(x)) \&
\hspace{1em} (\text{exit}(x) \rightarrow \exists P \text{ enter}(x)) \&
\hspace{1em} (\textbf{forall} y . (\text{exit}(y) \& P (\text{enter}(y) \& \exists P \text{ enter}(x))) \rightarrow \exists 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>


Pros

• Compact.
  • With $k$ bits we can represent $2^k$ values
  • $V$ values can be represented by $\log_2(V)$ bits

• We expect to pay little for "surplus" bits.

• We can extend the BDDs with additional bits dynamically if needed.

• Complementation is efficient (just switching the 0 and 1 leaves).

• Values not yet seen are represented by unused bit patterns (avoid using all bit patterns).
Cons

• We cannot compare variables beyond equality
• We cannot perform computations on values

**Prop allAnswersOk :**

\[
\text{forall } t_2 \ . \ \text{forall } a \ . \ \\
\text{answer}(t_2, a) \rightarrow \\
\text{exists } t_1 . \ \text{exists } q . \\
P \ \text{question}(t_1, q) \land \\
t_1 < t_2 \land \text{rightAnswer}(q) = a
\]
TraceContract

An internal Scala DSL for monitoring
generation of logs

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
**CommandMustSucceed:**

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

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

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

**excellent**
- I read the manual and was up and 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

DejaVu LogScope

DSL

parser

programming language

external DSL

TraceContract

DSL

programming language

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 for non-Scala programmers
- limited analyzability (for shallow internal DSLs)
Modeling in Scala: 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
- A "better Java"
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] {
    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
    }
}
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] {
    require {
        case Command(n, x) =>
            hot {
                case Fail(`n`, `x`) => error
                case Success(`n`, `x`) => ok
            }
    }
}
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] {
  require {
    case Command(n, x) =>
      not(Fail(n, x)) until (Success(n, x))
  }
}
monitor CommandMustSucceed {
    always {
        Command(n,x) => RequireSuccess(n,x)
    }
}

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

class ACommandMustSucceed extends Monitor[Event] {
    property {
        globally {
            Command("A",42) implies not(Fail("A", 42)) until (Success("A", 42))
        }
    }
}
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] {
  var count = 0
  require {
    case Command(n, x) if count < 10 =>
      count += 1
      not(Fail(n, x)) until (Success(n, x))
  }
}

first 10 commands must succeed
the **state** function

**CommandMustSucceed:**

“An issued command can succeed at most once”.

```java
class MaxOneSuccess extends Monitor[Event]
{
    require {
        case Success(_, number) =>
            state {
                case Success(_, `number`) => error
            }
    }
}
```
class TWTA_Ka extends Monitor[Event] {
    property { Init }

    def Init: Formula =
        state {
            case Command("TURNON", "TWTA", time, _) => On(time)
            case Command("TURNON", "KA", _, _) => error
        }

    def On(time: Int): Formula =
        state {
            case Command("TURNOFF", "TWTA", _, _) => Init
            case Command("TURNON", "KA", kaTime, _)
                if (time, kaTime) within (300 seconds) => error
        }
    }
}
rule-based system
for expressing past time logic

Success Has a Reason:

“A command success must be caused by an issued command”.

class SuccessHasAReson 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) ?-
    }
}
analyzing a trace

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

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

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

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)
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 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.

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 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":

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

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).
IMPLEMENTATION

how does it work?
formulas

```
abstract class Formula {
    def apply(event: Event): Formula
    def reduce(): Formula = this
    ...
}
```
basic formulas (single time point)

```
case object True extends Formula {
  override def apply(event: Event): Formula = this
}

case class Now(expectation: Event) extends Formula {
  override def apply(event: Event): Formula =
    if (expectation == event) True else False
}

... not(Fail(n, x)) until (Success(n, x)) ...

implicit def Event2Formula(event: Event): Formula = Now(event)
```
and

```scala
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
    }
  }
}
```
Until

\[ f_1 \mathbin{\mathcal{U}} f_2 = f_2 \lor \left( f_1 \land \mathcal{O}(f_1 \mathbin{\mathcal{U}} f_2) \right) \]

case class Until(formula1: Formula, formula2: Formula) extends Formula {
  override def apply(event: Event): Formula =
    Or(formula2(event), And(formula1(event), this).reduce()).reduce()
}
case class State(block: Block) extends Formula {
  override def apply(event: Event): Formula =
    if (block.isDefinedAt(event)) block(event) else this
}

// Hot the same

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

// Strong the same
at the end

def end(formula: Formula): Boolean =
formula match {
    case State(_) => true
    case Hot(_)   => false
    case Weak(_)  => true
    case Strong(_) => false
    case Until(_,_) => false
    case And(formula1, formula2) => end(formula1) && end(formula2)
    ...
}

observations

• high expressive power, easy to develop
• hard to analyze, learning curve for non-Scala programmers
THANKS!