Towards Combined Static and Runtime Verification of Distributed Software

Wolfgang Ahrendt

Chalmers University of Technology, Gothenburg, Sweden

Seminar on Distributed Runtime Verification
Bertinoro, 18 May 2016
### My Profile

<table>
<thead>
<tr>
<th>verification</th>
<th>static</th>
<th>combined static/runtime</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>deductive verif. of oo KeY</td>
<td><strong>STARVOORS</strong></td>
<td>runtime verif. of oo <strong>LARVA</strong></td>
</tr>
<tr>
<td>distributed</td>
<td>compositional ded. verif. of distributed oo</td>
<td>my next project (w. Schneider, Pace)</td>
<td>asked Borzoo ?to organise this seminar</td>
</tr>
</tbody>
</table>

Wolfgang Ahrendt  
Bertinoro 2016
<table>
<thead>
<tr>
<th>verification</th>
<th>static</th>
<th>combined</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>deductive verif. of oo</td>
<td><strong>STARVOORS</strong></td>
<td>runtime verif. of oo</td>
</tr>
<tr>
<td>applications</td>
<td>KeY</td>
<td></td>
<td><strong>LARVA</strong></td>
</tr>
<tr>
<td>distributed</td>
<td>compositional ded. verif.</td>
<td>my next project</td>
<td></td>
</tr>
<tr>
<td>applications</td>
<td>of distributed oo</td>
<td>(w. Schneider, Pace)</td>
<td></td>
</tr>
</tbody>
</table>
KeY

- Theorem prover for source code verification
- Functional (data-heavy) properties
- First-order *dynamic logic* as program logic
- Deductive verification, using a sequent calculus
- Verification = *symbolic execution* + induction/invariants
- Prover is *interactive* + highly automated
- Most elaborate KeY instance *KeY-Java*
  - Java as target language
  - Java dynamic logic
  - Supports specification language *JML*
Supported Specification Language:
JML

**Java Modeling Language**
Specification language for Java

International community effort lead by Gary T. Leavens, Iowa State
public static int max(int[] a) {
    int max = a[0], i = 1;
    while (i < a.length) {
        if (a[i] > max) max = a[i];
        ++i;
    }
    return max;
}
a != null
->
<

int max = 0;
if ( a.length > 0 ) max = a[0];
int i = 1;
while ( i < a.length ) {
    if ( a[i] > max ) max = a[i];
    ++i;
}
>
\forall int j; (j >= 0 & j < a.length -> max >= a[j])
&
(a.length > 0 ->
\exists int j; (j >= 0 & j < a.length & max = a[j]))
The KeY Verification System

- Program
- Specification

Proof Obligation Generator

DL Formula

Theorem Prover

- ✔️
- ❓
- ❌

- Long-term collaboration: Karlsruhe–Darmstadt–Chalmers
References

- Ahrendt, Beckert, Bubel, Hähnle, Schmitt, Ulbrich
  *The KeY Book*
  Springer, LNCS, to appear
Timsort

Hybrid sorting algorithm (insertion sort + merge sort) optimised for partially sorted arrays (typical for real-world data).

Facts

- Designed by Tim Peters (for Python)
- Since Java 1.7 default algorithm for non-primitive arrays/collections
- `java.util.Collections.sort` and `java.util.Arrays.sort` implement Timsort
Major Case Study with KeY

Verification with KeY

- Attempt to verify OpenJDK implementation
- 460 lines of specification vs. 928 lines of code
- KeY verification revealed bug
- Fixed version formally verified
- whole project 2.5 person month
Aftermath

▶ Bug affected
  ▶ Java (OpenJDK + Oracle)
  ▶ Android
  ▶ Python
  ▶ Apache: Lucene, Hadoop, Spark++
  ▶ Go, D, Haskell

▶ blog with >3 million page views
▶ top news on ycombinator, reddit, Hacker News etc.
References

## Part II

<table>
<thead>
<tr>
<th>verification</th>
<th>static</th>
<th>combined static/runtime</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>deductive</td>
<td>STAARVOORS</td>
<td>runtime</td>
</tr>
<tr>
<td>applications</td>
<td>verif. of oo KeY</td>
<td></td>
<td>LARVA</td>
</tr>
<tr>
<td>distributed</td>
<td>compositional</td>
<td>my next project</td>
<td></td>
</tr>
<tr>
<td>applications</td>
<td>ded. verif. of</td>
<td>(w. Schneider, Pace)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>distributed oo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Logical Automata for Runtime Verification and Analysis

Property language:

DATE (Dynamic Automata w. Timers and Events)

Front ends: Duration Calculus, Lustre

Monitor generation + Code instrumentation w. Aspects
- communicating automata, event-triggered transitions, timers
- events: method entry/exit, timer events, synchronising events

```
foreach transfer:
```

```
start \rightarrow \text{connDrop}^{\dagger} c \geq 5 \text{ unreliable!}
```

```
start \rightarrow \text{connDrop}^{\dagger} c < 5 c ++
```

```
start \rightarrow \text{start}^{\dagger}(transfer) \\|
unreliable? \\|
receive^{\dagger} \\|
end^{\dagger}(transfer) \\|
```

```
receive^{\dagger} \\|
```

```
bad
```

Wolfgang Ahrendt
Bertinoro 2016
References

- C. Colombo, G.J. Pace, G. Schneider  
  *LARVA – A Tool for Runtime Monitoring of Java Programs*  
  SEFM’09, 2009. IEEE Computer Society
<table>
<thead>
<tr>
<th>verification</th>
<th>static</th>
<th>combined static/runtime</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential applications</td>
<td>deductive verif. of oo KeY</td>
<td><strong>STARVOOrS</strong></td>
<td>runtime verif. of oo <strong>LARVA</strong></td>
</tr>
<tr>
<td>distributed applications</td>
<td>compositional ded. verif. of distributed oo</td>
<td>my next project (w. Schneider, Pace)</td>
<td></td>
</tr>
</tbody>
</table>
Combined Static and Runtime Verification: **STARVOORS**

\[ ppDATE : \]

- Specification Language for Data and Control Properties
- Extends DATE with
  - pre/post-conditions, associated to the automata’s states:

\[
q_1 \xrightarrow{e|cond\rightarrow act} q_2
\]

\[ \tau(q_1) = \{ \ldots, \{\text{pre}\} \text{ m } \{\text{post}\}, \ldots \} \]
**Example**:
Scenario including a HasTable

\[
\tau(q_1) = \{ \{ \text{size} < \text{capacity} \} \text{ add}(o) \{ \exists \ i. \text{arr}[i] = o \} \}
\]

- Hoare triples are described using JML-like notation
High-level description of the framework
High-level description of the framework

Static

- Partial Specification Evaluation
- ppDATE
- Specification Translation
- DATE
- Monitor
- Generator
- Aspects
- Code Instrumentation
- Weaving Code
- (partial) Proofs
- Deductive Verifier
- ppDATE
- S'
KeY tries to prove:
\{size < capacity\} \text{add}(o) \{\exists \, i\, . \, \text{arr}[i] = o\}

KeY cannot fully prove (automatically)

proof branch
\ldots, \text{arr}[key\mod capacity] = \text{null} \vdash \ldots
closed (automatically)

proof branch
\ldots, \neg \text{arr}[key\mod capacity] = \text{null} \vdash \ldots
not closed (automatically)
High-level description of the framework
Partial proof analysis synthesises additional pre-conditions, here

\[ \neg \text{arr[key\%capacity]} = \text{null} \]

\[ \tau(q_1) = \]
\[
\{ \{\text{pre}\} \text{add(o)} \{\text{post}\} \} \]
Partial proof analysis synthesises additional pre-conditions, here

\[ \neg \text{arr[key\%capacity]} = \text{null} \]

\[
\tau(q_1) = \\
\{ \{\text{pre} \land \neg \text{arr[key\%capacity]} = \text{null}\} \text{add(o)} \{\text{post}\} \}
\]
**StaRVOORS: Mondex Case Study**

<table>
<thead>
<tr>
<th>Transactions</th>
<th>no monitoring</th>
<th>monitoring without static verif.</th>
<th>monitoring using static verif.</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8 ms</td>
<td>120 ms</td>
<td>15 ms</td>
</tr>
<tr>
<td>100</td>
<td>50 ms</td>
<td>3,500 ms</td>
<td>90 ms</td>
</tr>
<tr>
<td>1000</td>
<td>250 ms</td>
<td>330,000 ms</td>
<td>375 ms</td>
</tr>
</tbody>
</table>

- **Main reason for overhead: postcondition monitors**
- **KeY proves 2 Hoare triples fully**
  - $\Rightarrow$ **never** checked at runtime
- **KeY proves 24 Hoare triples partially**
  - $\Rightarrow$ **conditionally** checked at runtime
References

- W. Ahrendt, G. Pace, G. Schneider
  *A Unified Approach for Static and Runtime Verification: – Framework and Applications*
  ISoLA 2012

- W. Ahrendt, M. Chimento, G. Pace, G. Schneider
  *A Specification Language for Static and Runtime Verification of Data and Control Properties*
  FM 2015

- W. Ahrendt, M. Chimento, G. Pace, G. Schneider
  *StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java*
  RV 2015
### Part IV

<table>
<thead>
<tr>
<th>verification</th>
<th>static</th>
<th>combined static/runtime</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential applications</td>
<td>deductive verif. of oo KeY</td>
<td><strong>STARVOORS</strong></td>
<td>runtime verif. of oo <strong>LARVA</strong></td>
</tr>
<tr>
<td>distributed applications</td>
<td>compositional ded. verif. of distributed oo</td>
<td>my next project (w. Schneider, Pace)</td>
<td></td>
</tr>
</tbody>
</table>

Wolfgang Ahrendt  
Bertinoro 2016
Here:

An object-oriented approach to distributed systems:

- asynchronous method calls
- incoming calls spawn object local threads
- shared memory is object local

Target languages:

1. Creol [Univ. Oslo]
2. ABS [HATS project]
Properties of messages and their data

Properties talk about communication history ($\mathcal{H}$)

Outer specification of interface ‘Bank’:

$$ b.\text{getBalance}(c) = \left( \sum_{b.\text{deposit}(c,x) \in \mathcal{H}} (x) \right) - \left( \sum_{b.\text{withdraw}(c,x) \in \mathcal{H}} (x) \right) $$

Inner specification of class ‘Swedbank implements Bank’:

$$ \text{getBalance}(c) = \text{accounts.get}(c).\text{balance} $$
(Deductive) Concurrent Verification

<table>
<thead>
<tr>
<th></th>
<th>shared memory</th>
<th>message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-</td>
<td>interference freedom tests</td>
<td>cooperation tests</td>
</tr>
<tr>
<td>compositional</td>
<td>[Owicky, Gries 75]</td>
<td>[Apt et al. 80]</td>
</tr>
<tr>
<td>compositional</td>
<td>rely-guarantee [Jones 81]</td>
<td>assumption-commitment [Misra, Chandy 81]</td>
</tr>
</tbody>
</table>

“assume-guarantee”

- coined by Abadi & Lamport for union of compositional approaches
- also refers to compositional model checking techniques (Pnueli, ..., Steffen, ..., Păsăreanu)
Assumption-Commitment

Component contract:
- *assumptions* about messages (+ data) *from* environment
- *commitments* about messages (+ data) *to* environment

Component internal invariant:
- refinement relation between external history and internal state

Component verification:
- **Sequential verification**, using assumptions showing commitments of this component
  showing assumptions using commitments of called components
Variation of AC:

- Specs don’t distinguish between assumption and commitment, but represent both in history invariant.
- Component has to guarantee that outgoing messages maintain invariant, given that incoming messages do.
Calculus: Object Communication

\[ \Gamma \vdash \{ U_{\mathcal{H}} \} inv(\mathcal{H})^{this,o}_{\text{caller}, callee} \quad \Gamma \vdash \{ U_{\mathcal{H}} \}[rest] \phi \]

\[ \Gamma \vdash [ \ l \ ! \ o.mtd(\overline{x}); \ rest] \phi \]

- **rest**: rest of the program
- **\mathcal{H}**: history
- **inv**: interface invariant
- **U_{\mathcal{H}}**: \( \mathcal{H} := \text{extend}(\mathcal{H}, \text{Invoc}(l, \overline{x})) \)
Calculus: Object Communication

\[ \Gamma \vdash \{ U_{\mathcal{H}, \bar{y}} \}[rest] \phi \]
\[ \Gamma \vdash [l?(\bar{y}) \text{; rest}] \phi \]

- \textit{inv}: interface invariant
- \textit{\mathcal{H}}: history
- \( U_{\mathcal{H}, \bar{y}} \): append completion message
  + assume invariant:
  \[ \mathcal{H}, \bar{y} := \text{some } H, \bar{v}.(\mathcal{H} \leq H \land \text{Comp}(H, l, \bar{v}) \land \text{inv}(H)_{\text{this, l.callee}}) \]
W. Ahrendt, M. Dylla
*A System for Compositional Verification of Asynchronous Objects*
Science of Computer Programming

C. Din, R. Bubel, R. Hähnle
*KeY-ABS: A deductive verification tool for the concurrent modelling language ABS*
CADE 2015
<table>
<thead>
<tr>
<th>verification</th>
<th>static</th>
<th>combined static/runtime</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential applications</td>
<td>deductive verif. of oo KeY</td>
<td><strong>STARVOORS</strong></td>
<td>runtime verif. of oo <strong>LARVA</strong></td>
</tr>
<tr>
<td>distributed applications</td>
<td>compositional ded. verif. of distributed oo</td>
<td><strong>my next project</strong> (w. Schneider, Pace)</td>
<td></td>
</tr>
</tbody>
</table>
Assume-Guarantee (AG) Runtime Verification?

AG mostly (or only?) used in static verification

Static AG has bottlenecks:
1. Verifying refinement of
   - communication contract to
     - inner state properties
   can require smart proof engineering.
2. It requires full access to implementation of all components. But in practice, some components are closed: proprietary components, legacy components, binaries.

Claim:
AG has great potential for combined static/runtime verification
Assume-Guarantee Static+Runtime Verification

Address bottlenecks of Assume-Guarantee:

For open components:

▶ *Attempt* static component verification,
  but refer ‘proof-holes’ to runtime verification

For closed components:

▶ formalise expected assumptions/guarantees ⇒ AG contract
▶ runtime verify compliance of *closed* components w. contract
▶ verify *open* components that use closed ones:
  using commitments
  showing assumptions
  of closed components they use
  (statically, or at runtime, or with combination)
References

- W. Ahrendt, G. Pace, G. Schneider
  *StaRVOOrS – Episode II,*
  *Strengthen and Distribute the Force*
  submitted