Unified Static and Runtime Verification of Object-Oriented Software

Wolfgang Ahrendt\textsuperscript{1},
Mauricio Chimento\textsuperscript{1}, Gerardo Schneider\textsuperscript{2}, Gordon J. Pace\textsuperscript{3}

\textsuperscript{1}Chalmers University of Technology, Gothenburg, Sweden
\textsuperscript{2}University of Gothenburg, Sweden
\textsuperscript{3}University of Malta

Tallinn, August 2014
Static Verification vs. Runtime Verification

- Static verification
  - High precision
  - Use abstractions for increased automation
  but
  - Powerful judgements hard to achieve automatically
  - Often losing aspects of concrete system

- Runtime verification
  - Full precision (including real deployment)
  - Full automation
  but
  - Cannot judge future runs
  - Computational overhead of monitoring the running system
Project on Unified Static and Runtime Verification

Unified Static and Runtime Verification of Object-Oriented SW

Members:

▶ Wolfgang Ahrendt,
  Chalmers University of Technology
▶ Mauricio Chimento,
  Chalmers University of Technology
▶ Gerardo Schneider,
  University of Gothenburg

External collaborator:

▶ Gordon J. Pace,
  University of Malta
Project on Unified Static and Runtime Verification

Unified Static and Runtime Verification of Object-Oriented SW

STARVOORS

Members:

- Wolfgang Ahrendt,
  Chalmers University of Technology
- Mauricio Chimento,
  Chalmers University of Technology
- Gerardo Schneider,
  University of Gothenburg

External collaborator:

- Gordon J. Pace,
  University of Malta
Framework for Unified Static and Runtime Verification

- Combine static and runtime verification
  - Combine data centric and control centric properties
  - Unified specification for both
- Use (partial) static verification results for partial evaluation of properties
- Runtime verification of resulting properties
- Increase safety and efficiency
LARVA: A Runtime Verification Tool for Java

LARVA ≡ Logical Automata for Runtime Verification and Analysis

- targets Java applications
- checks control oriented properties (untimed and real-time), specified in
  - DATE (Dynamic Automata with Timers and Events)
  - Lustre
  - duration calculus
DATE Automaton Example

```
connDrop \ c == 5 \ unreliable!
```

```
connDrop \ c < 5 \ c ++
```

In general:

- communicating automata, event-triggered transitions, timers
- events: method entry/exit, timer events, synchronising events
DATE Automaton Example

\[ \text{connDrop} \uparrow c = 5 \text{ unreliable!} \]

\[ \text{connDrop} \uparrow c < 5 \text{ ++} \]

\textit{foreach} transfer :

\[ \text{start} \uparrow (\text{transfer}) \]

\[ \text{unreliable?} \]

\[ \text{receive} \uparrow \]

\[ \text{bad} \]
DATE Automaton Example

In general:

- communicating automata, event-triggered transitions, timers
- events: method entry/exit, timer events, synchronising events
**LARVA Functionality**

- **LARVA input**
  - DATE automaton (or alternative format)
  - application code
**LARVA Functionality**

- **LARVA input**
  - DATE automaton (or alternative format)
  - application code

- **LARVA output**
  - monitor
  - instrumented application code, with triggers for monitor
KeY is an approach and tool for the
- Formal specification of foremost functional properties
- Deductive verification, i.e., using theorem proving of
- OO software, foremost Java and ABS
KeY

- Dynamic logic (generalisation of Hoare logic) as program logic
- Verification = symbolic execution + induction/invariants
- Sequent calculus
- Prover is automated + interactive
- most elaborate KeY instance KeY-Java
  - Java as target language
  - Supports specification language JML
ppDATE:

- Extending DATE with pre/post-conditions, associated to the automata’s states:

\[ q \xrightarrow{\text{event}|\text{cond} \rightarrow \text{act}} q' \]

\[ \tau(q) = \{ \ldots, \{\text{pre}\} \text{ method } \{\text{post}\}, \ldots \} \]

- Transition enabled if \textit{cond} holds
ppDATE trace $w \in (\Sigma^\updownarrow \times \Theta)^*$ is violating prefix if either

1. $(q_0, v_0) w_1 = \Rightarrow (q, v)$
2. $\tau(q) \ni \{\text{pre}\} m \{\text{post}\}$
3. $\theta_1 \mid = \text{pre}$
4. $\theta_2 \not\mid = \text{post}$
ppDATE trace $w \in (\Sigma^{\uparrow} \times \Theta)^*$ is violating prefix if either

- $<(q_0, v_0)>^w (q, v)$ and $q \in BadStates$
Violating Traces

A violating trace has a violating prefix if either

\[ (q_0, v_0) \xrightarrow{w} (q, v) \] and \( q \in \text{BadStates} \)

\[ w = w_1 + \langle (m_{id}, \theta_1) \rangle + w_2 + \langle (m_{id}, \theta_2) \rangle \]

such that:

1. \( (q_0, v_0) \xrightarrow{w_1} (q, v) \)
2. \( \tau(q) \ni \{ \text{pre} \} m \{ \text{post} \} \)
3. \( \theta_1 \models \text{pre} \)
4. \( \theta_2 \not\models \text{post} \)
Violating Traces

A violating trace has a violating prefix

\[ \text{ppDATE trace } w \in (\Sigma \downarrow \times \Theta)^* \text{ is violating prefix if either} \]

\[ (q_0, v_0) \xrightarrow{w} (q, v) \text{ and } q \in \text{BadStates} \]

\[ w = w_1 + \langle m_{id} \downarrow, \theta_1 \rangle + w_2 + \langle m_{id} \uparrow, \theta_2 \rangle \]

such that:

1. \( (q_0, v_0) \xrightarrow{w_1} (q, v) \)
2. \( \tau(q) \ni \{\text{pre}\} m \{\text{post}\} \)
3. \( \theta_1 \models \text{pre} \)
4. \( \theta_2 \not\models \text{post} \)
High-level description of the framework
Case study: Login Example

Scenario:

- At login, new users are added to set users
- Assume users is implemented using hashing with open addressing
- Adding implemented by users.add(u,key)
Case study: Login Example

\[
\tau(q) = \{ \{ \text{users.size} < \text{users.capacity} \} \text{ add } \{ \text{post} \} \}
\]

\[
\text{post} \equiv (\exists \text{ int } i; i \geq 0 \land i < \text{users.capacity}; \text{users.h}[i] = o;)
\]
Translation of Hoare triple to JML

```java
class HashTable {
    ...

    /**
     * @ public normal_behavior
     *  @ requires size < capacity;
     *  @ ensures (\exists int i;
     *           i>= 0 && i < capacity;
     *           h[i] == o);
     *  @ assignable size, h[*];
     ...
     */

    public void add (Object o, int key) {}
public void add (Object o, int key) {
    ...
    int i = hash(key);
    if (h[i] == null) {
        h[i] = o; size++;
    }
    else {
        while ... \ store at next free slot
        ...
    }
}
KeY tries to prove:
size < capacity → ⟨add(o, key)⟩ ∃i. h[i] = o

KeY will produce branches:
..., h[key%capacity] = null ⊢ ...
and
..., ¬h[key%capacity] = null ⊢ ...

first branch closes automatically, the second doesn’t
First, for $\tau(q)$ replace $\{pre\}$ add $\{post\}$ by
First, for $\tau(q)$ replace \{pre\} add \{post\} by
\[
\{\text{pre} \wedge \neg \text{users.h[key\%capacity]} = \text{null}\} \text{ add } \{\text{post}\}
\]
and
\[
\{\text{pre} \wedge \text{users.h[key\%capacity]} = \text{null}\} \text{ add } \{\text{true}\}
Second, new argument is added to distinguish different calls.
Second, new argument is added to distinguish different calls

```java
public void add (Object o, int key) {
    addAux(fid.getNewId(), o, key);
}

public void addAux (Integer id, Object o, int key) {
    //same code as add had before.
}
```

```plaintext
{pre ∧ ¬users.h[key%capacity] = null} addAux {post}
and

{pre ∧ users.h[key%capacity] = null} addAux {true}
```
Case study: Login Example - Model Transformation

$q \xrightarrow{\text{addAux}_id | \text{pre} \rightarrow \text{s}_id!} q'$

Graph:
- Start state
- Transition: $s_id?$
- Transition: $\text{addAux}_id | \text{users.opPost}()$
- Transition: $\text{addAux}_id | \neg\text{users.opPost}()$
- Final states: $ok$, $bad$
Case study: Login Example - Model Transformation

$q_{\text{addAux}_{id} \uparrow | \text{users.contains}(o, \text{key}) = true \rightarrow \text{if pre then } s_{\text{id}}!} \rightarrow q'$

Diagram:

- Start state
- Transition labeled $s_{\text{id}}?\rightarrow\text{addAux}_{id} \uparrow | \text{users.opPost()}$
- Transition labeled $\text{addAux}_{id} \uparrow | \neg \text{users.opPost()}$
- End states:
  - Ok
  - Bad

Formulas:

- $\text{addAux}_{id} \uparrow | \neg (\text{users.contains}(o, \text{key}) = \text{true}) \land (\text{pre} \land \neg \text{users.h}[\text{key} \% \text{capacity}] = \text{null}) \rightarrow s_{\text{id}}! \rightarrow q$
Finally, LARVA generates the monitors which will control the partially verified property.
Wolfgang Ahrendt, Gordon J. Pace, Gerardo Schneider
A Unified Approach for Static and Runtime Verification:
– Framework and Applications
ISoLA 2012
Springer, LNCS 7609