A Comparison of Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems

Crystal Chang Din\textsuperscript{1}, Richard Bubel\textsuperscript{2}, Olaf Owe\textsuperscript{1}

University of Oslo, Norway\textsuperscript{1},
Technische Universität Darmstadt, Germany\textsuperscript{2}

25th Nordic Workshop on Programming Theory
Tallinn, Estonia, 20 Nov.-22 Nov. 2013
Introduction

- Concurrent and distributed systems
- Object orientation
- Asynchronous method calls
- Futures
- ABS: Abstract Behavior Specification language
Purpose of the talk

- **Focus:**
  - Futures

- **Challenge:**
  - Goal: local reasoning
  - Futures are global entities shared between objects

- **Earlier work:**
  - A sound compositional reasoning system for ABS with futures

- **This work:**
  - Experience with tool support:
    - [Dynamic] runtime assertion checking (RAC) vs. [Static] theorem proving (KeY)
The Syntax and Semantics of Asynchronous Method Calls and Futures

A *blocking* method call:
\[ Fut < Int > u := o'!m(); \text{Int x := u.get;} \]

A *non-blocking* method call:
\[ Fut < Int > u := o'!m(); \text{await u?; \text{Int x := u.get;}} \]

4-event (invoc, invocR, futr, fetch) semantics of method calls:

```
\begin{align*}
&<o \rightarrow o', u, m, e> \\
&<o' \rightarrow o, u, m, e> \\
&<o' \leftarrow o', u, m, e> \\
&<o'' \leftarrow o'', u, e>
\end{align*}
```
class Service(Int limit) implements ServiceI{
    ProxyI proxy; ProxyI lastProxy;

    {proxy := new Proxy(limit, this);
     lastProxy := proxy; this!produce();}

    Void subscribe(ClientI cl){
        Fut<ProxyI> last := lastProxy!add(cl);
        lastProxy := last.get;}

    Void produce(){
        Fut<News> fut := prod!detectNews();
        proxy!publish(fut); Delegation to proxy! No blocking!
    }
}
Runtime Assertion Checking for Pub-Sub Example

Data Types of Events in ABS:

```
data Any = O | F | AR | any;
data Event =
    Invoc(Any o, Any f, String m, Any a) |
    InvocR(Any o, Any f, String m, Any a) | ... ;
```
Data Types of Events in ABS:

data Any = O | F | AR | any ;

data Event =
    Invoc(Any o, Any f, String m, Any a) |
    InvocR(Any o, Any f, String m, Any a) | ... ;

Invariant for Service Class:

has(Invoc(any,any,"add",any)) =>
    isSubseq(list[Invoc(any,any,"add",AR),
                  InvocR(any,any,subscribe",AR)])

Pattern Matching
Pattern matching vs. quantification

In *runtime assertion checking* using Maude we can use pattern matching

\[
\text{has (InvocEv (any, any, "add", any))}
\]

to express that we are only interested in invocations of method *add*. 
Pattern matching vs. quantification

In runtime assertion checking using Maude we can use pattern matching

\[
\text{has (InvocEv}(\text{any, any, "add", any) \text{)}
\]

to express that we are only interested in invocations of method \text{add}.

On the logic level (for verification), we need to use quantifiers

\[
\exists a, b, c. (\text{invocEv}(a, b, \text{add}, c) \in \text{history})
\]

\[\Rightarrow \text{negative impact on automation}\]
Theorem Proving for Pub-Sub Example

- Unspecified sequence of local events extended at process release points
- Quantifiers in theorem proving
- Additional invariants expressing what COULD NOT have happened while process is released
Closed or Open Systems

Open vs. Closed System Assumption

In case of the *closed* system assumption, we assume to have complete knowledge about the system. While when using the *open* system assumption, we do not assume that we know every type and method implementation.
Closed or Open Systems

Open vs. Closed System Assumption

In case of the closed system assumption, we assume to have complete knowledge about the system. While when using the open system assumption, we do not assume that we know every type and method implementation.

Example (Call sequence)

Assume the following invocations of \( m \) and \( n \) are the only invocations of these method in the current system implementation:

\[
\text{Fut<Int> } r = o!m(); \quad \text{Fut<Int> } l = o!n(); \ldots
\]

Closed system: Exploitation of the implicit knowledge that a method invocation of \( m \) is always followed by a method invocation event of \( n \) ⇒ simpler invariant specifications (e.g., RAC)
Closed or Open Systems

Open vs. Closed System Assumption

In case of the *closed* system assumption, we assume to have complete knowledge about the system. While when using the *open* system assumption, we do not assume that we know every type and method implementation.

Example (Call sequence)

Assume the following invocations of \( m \) and \( n \) are the only invocations of these method in the current system implementation:

\[
\text{Fut}\langle\text{Int}\rangle \ r = o!m(); \quad \text{Fut}\langle\text{Int}\rangle \ l = o!n(); \quad \ldots
\]

*Open system*: Exploitation of the implicit knowledge not possible (as \( m, n \) might be called differently somewhere else). If required, needs to be specified explicitly \( \Rightarrow \) more complex specifications needed for open systems (here: verification)
ABS Method Annotations

**Require**: $\text{pre}_m$  
**Ensure**: $\text{post}_m \land \text{inv}_m$

Unit $m()$ {
  ... \textbf{assert} $\text{inv}_m$; \textbf{await} $b$; ...\}
Unit $n()$ {
  ... $o!m();$ ...\}

\[\]

Crystal Chang Din @ UiO  
Runtime assertion checking vs. Theorem proving  
NWPT'13
ABS Method Annotations

[Require: $pre_m$] [Ensure: $post_m \&\& inv_m$]

Unit $m()$ { ... assert $inv_m$; await $b$; ...}
Unit $n()$ { ... $o!m$(); ...}

Execution Order:
ABS Method Annotations

[Require: \( \text{pre}_m \)] [Ensure: \( \text{post}_m \) \&\& \( \text{inv}_m \)]

Unit \( m() \) { ... \text{assert} \ \text{inv}_m; \ \text{await} \ b; \ ... \}
Unit \( n() \) { ... \text{o}!m(); \ ... \}

Execution Order:
invocr
ABS Method Annotations

[Require: \texttt{pre}_m] [Ensure: \texttt{post}_m \land \texttt{inv}_m]

Unit \texttt{m}() \{ \ldots \texttt{assert inv}_m; \texttt{await} b; \ldots \}
Unit \texttt{n}() \{ \ldots \texttt{o!m}(); \ldots \}

Execution Order:
\texttt{invocr} \quad \texttt{Require}
ABS Method Annotations

**[Require: pre}_m**] [**Ensure: post}_m & & inv}_m**]

Unit m() { ... **assert inv}_m; await b; ... }
Unit n() { ... o!m(); ... }

Execution Order:
invocr Require methodBody
ABS Method Annotations

[Require: $\text{pre}_m$]  [Ensure: $\text{post}_m \land \text{inv}_m$]

Unit $m()$ { ... assert $\text{inv}_m$; await $b$; ... }
Unit $n()$ { ... o!$m()$; ... }

Execution Order:
invocr  Require  methodBody  futr
ABS Method Annotations

[Require: $pre_m$] [Ensure: $post_m \&\& inv_m$]

Unit $m()$ { ... assert $inv_m$; await $b$; ... }
Unit $n()$ { ... o!m(); ... }

Execution Order:
invocr Require methodBody futr Ensure
Applicability of pre- and post condition

Gap between method invocation and actual execution.

- A property holds at invocation time, but
- It might no longer be at time of invocation reaction.
Experience from the Example (1)

- Verification:
  - semi-automation

- RAC:
  - invariants for service/proxy not client/producer
  - only use invariants but no pre- and postconditions
  - pattern matching is needed
  - execution from different initial states
  - consistent with the expected result
Reader Writer Example

class RWController implements RWinterface{
    CallerI writer := null;

    Void openW(){
        await writer = null; writer := caller;
    }

    Void closeW(){
        await writer = caller; writer := null;
    }
    ...
}
Reader Writer Example

class RWController implements RWInterface{
    CallerI writer := null;

    Void openW(){
        await writer = null;  writer := caller;}

    Void closeW(){
        await writer = caller;  writer := null;}

    ...}

A state-based history properties for the RWController class:

\[ \text{inv}_{\text{RWController}} \triangleq \text{Writers}(\mathcal{H}) = \{\text{writer}\} - \text{null} \]
Implementation of the History Functions for RAC

The definition of *Writers* : $\text{Seq}[\text{Ev}] \rightarrow \text{Set}[\text{Obj}]$ is:

- $\text{Writers}(\text{Nil}) \triangleq \text{EmptySet}$
- $\text{Writers}(h \cdot \langle \leftarrow \text{this}, fr', \text{openW}, _\rangle) \triangleq \text{Writers}(h) + \text{irev}(h, fr').\text{caller}$
- $\text{Writers}(h \cdot \langle \leftarrow \text{this}, fr', \text{closeW}, _\rangle) \triangleq \text{Writers}(h) - \text{irev}(h, fr').\text{caller}$
- $\text{Writers}(h \cdot \text{others}) \triangleq \text{Writers}(h)$
Theorem Proving for Reader Writer Example

The formulation of $\text{inv}_{\text{RWController}}$:

$$\text{length(getWriters(h))} \leq 1 \land \text{self.writer} =$$

$$(\text{length(getWriters(h))} = 0 \ ? \ \text{null} : \text{getWriters(h).get(0)})$$
Theorem Proving for Reader Writer Example

The formulation of $\text{inv}_{\text{RWController}}$:

\[
\text{length}(\text{getWriters}(h)) \leq 1 \land \text{self.writer} = \\
(\text{length}(\text{getWriters}(h)) = 0 \land \text{null}: \text{getWriters}(h).\text{get}(0))
\]

The corresponding axiomatization in our dynamic logic:

\[
\forall h \forall w (\quad w \neq \text{null} \land \\
\exists i (\text{getWriters}(h).\text{get}(i) = w) \\
\Leftrightarrow \\
\exists e (\text{isFutEv}(e) \land e \in h \land \\
\text{getMethod}(e) = \text{openW} \land \\
w = \text{getCaller}(\text{getIREv}(h, \text{getFut}(e))) \land \\
\forall e' (\text{later}(e', e, h) \land \text{isFutEv}(e') \rightarrow \text{getMethod}(e') \neq \text{closeW}))
\]

Crystal Chang Din @ UiO

Runtime assertion checking vs. Theorem proving

NWPT’13 15 / 17
Experience from the Example (2)

- Verification:
  - semi-automation which is better than the case for pub-sub example

- RAC:
  - inductive definition for history functions
  - execution from different initial states
  - easy to show fairness
  - consistent with the expected result
Contribution

1. Implementation of a runtime assertion checker for ABS programs
   - Explicit representation of the global history
   - Support for inline method annotations to specify behavioral properties
   - Specification constructs of state- and trace-based history properties

2. Extension of the KeY theorem prover for ABS programs
   - Specification constructs of state- and trace-based history properties
   - Implementation of extra rules for handling history properties

3. Comparing history-based checking techniques with their sequential counterpart
   - RAC is nearly on par with that of a sequential setting
   - Deductive verification is harder because the support for reasoning about histories is not yet automatized to a high degree.