State of art and challenges in deductive verification of concurrent programs with cooperative scheduling

Crystal Chang Din
Software Engineering Group, TU Darmstadt, Germany

28.09.2015

http://www.envisage-project.eu
The concurrency model of the ABS language

- ABS stands for Abstract Behavioral Specification
- Active objects
- Futures
- Cooperative scheduling
  - scheduling points are made syntactically explicit in the code
  - `await` statements (on boolean expressions or futures)
An ABS example

```java
class C() implements CI {
    Int x = 0;

    Unit m(CI o){
        await x > 0;
        /* a sequence of statements which does not contain release points */
        Fut<Bool> f = o!n(...);
        await f??;
        /* a sequence of statements which does not contain release points */
    }

    Bool n(){...}
}
```
Research Question 1

How to achieve compositional reasoning for ABS concurrent models?
Disjoint-event semantics for ABS method calls
Disjoint-event semantics for ABS method calls

\[ \text{invocEv}(o, o', fr, m, e) \]

\[ \text{invocREv}(o, o', fr, m, e) \]

\[ \text{compEv}(o', fr, m, e) \]

\[ \text{compREv}(o'', fr, e) \]
The data structure of histories

- **Sequence** is the data structure of histories!
- For example:
  \[ H := \text{concat}(H, \text{invocEv}(o, o', fr, m, \bar{e})), \]
  where \( H \) is a history sequence
Class invariant

- An invariant of a class $C$ specifies invariant of instances of $C$.

- The class invariant serves as a contract between the different processes of the object
  - must hold
    - after initialization
    - after method termination
    - before suspension (on `await` statement)

- may assume
  - when method starts
  - after suspension (on `await` statement)
Class invariant

- establishes a relationship between the internal state and the observable behavior of the class instances
- defines the observable behavior in terms of the structure of communication histories
Network-on-Chips (NoC)
packet switching platform
class Router(Pos address, Int buffSize) implements RouterI {
    Ports ports = EmptyMap;
    Set<Packet> receivedPks = EmptySet;

    Unit setPorts(Router e, Router w, Router n, Router s){
        ports = map[Pair(N, P(True, True, n, 0)), Pair(S, P(True, True, s, 0)),
                Pair(E, P(True, True, e, 0)), Pair(W, P(True, True, w, 0))];
    }

    Unit getPk(Packet pk, Direction srcPort){...

    Unit redirectPk(Packet pk, Direction srcPort){...

}
The redirectPk method

Unit redirectPk(Packet pk, Direction srcPort) {
    Direction direc = xFirstRouting(addressPk(pk), address);
    await (inState(lookup(ports, srcPort)) == True) &&
        (outState(lookup(ports, direc)) == True);

    ports = put(ports, srcPort,
                inSet(lookup(ports, srcPort), False));
    ports = put(ports, direc,
                outSet(lookup(ports, direc), False));

    Router r = rId(lookup(ports, direc));
    Fut<Unit> f = r!getPk(pk, opposite(direc));
    await f?;

    ports = put(ports, srcPort,
                decreaseBuff(lookup(ports, srcPort)));
    ports = put(ports, srcPort,
                inSet(lookup(ports, srcPort), True));
    ports = put(ports, direc,
                outSet(lookup(ports, direc), True));
}
The redirectPk method

Unit redirectPk(Packet pk, Direction srcPort){
    Direction direc = xFirstRouting(addressPk(pk), address);
    await (inState(lookup(ports,srcPort)) == True) &&
            (outState(lookup(ports,direc)) == True);
    ports = put(ports, srcPort,
                inSet(lookup(ports, srcPort), False));
    ports = put(ports, direc,
                outSet(lookup(ports, direc), False));
    Router r = rId(lookup(ports, direc));
    Fut<Unit> f = rlgetPk(pk, opposite(direc));
    await f?;
    ports = put(ports, srcPort,
                decreaseBuff(lookup(ports, srcPort)));
    ports = put(ports, srcPort,
                inSet(lookup(ports, srcPort), True));
    ports = put(ports, direc,
                outSet(lookup(ports, direc), True));
}
The redirectPk method

Unit redirectPk(Packet pk, Direction srcPort){
    Direction direc = xFirstRouting(addressPk(pk), address);
    await (inState(lookup(ports, srcPort)) == True) &&
        (outState(lookup(ports, direc)) == True);
    ports = put(ports, srcPort,
        inSet(lookup(ports, srcPort), False));
    ports = put(ports, direc,
        outSet(lookup(ports, direc), False));
    Router r = rId(lookup(ports, direc));
    Fut<Unit> f = r!getPk(pk, opposite(direc));
    await f?;
    ports = put(ports, srcPort,
        decreaseBuff(lookup(ports, srcPort)));
    ports = put(ports, srcPort,
        inSet(lookup(ports, srcPort), True));
    ports = put(ports, direc,
        outSet(lookup(ports, direc), True));
}

← invocREv(_, this, fr, redirectPk, (pk, srcP))
← check if input and output channels are available
The redirectPk method

Unit redirectPk(Packet pk, Direction srcPort){
    Direction direc = xFirstRouting(addressPk(pk), address);
    await (inState(lookup(ports,srcPort)) == True) &&
        (outState(lookup(ports,direc)) == True);

    ports = put(ports, srcPort,
                inSet(lookup(ports, srcPort), False));
    ports = put(ports, direc,
                outSet(lookup(ports, direc), False));

    Router r = rId(lookup(ports, direc));
    Fut<Unit> f = r!getPk(pk, opposite(direc));
    await f?;

    ports = put(ports, srcPort,
                decreaseBuff(lookup(ports, srcPort)));
    ports = put(ports, srcPort,
                inSet(lookup(ports, srcPort), True));
    ports = put(ports, direc,
                outSet(lookup(ports, direc), True));
}

← invocREV(_, this, fr, redirectPk, (pk, srcP))

← check if input and output channels are available

← block input and output channels
The redirectPk method

Unit redirectPk(Packet pk, Direction srcPort) {
    Direction direc = xFirstRouting(addressPk(pk), address);
    await (inState(lookup(ports, srcPort)) == True) &&
        (outState(lookup(ports, direc)) == True);

    ports = put(ports, srcPort, inSet(lookup(ports, srcPort), False));
    ports = put(ports, direc, outSet(lookup(ports, direc), False));

    Router r = rId(lookup(ports, direc));
    Fut<Unit> f = r!getPk(pk, opposite(direc));
    await f?;

    ports = put(ports, srcPort, decreaseBuff(lookup(ports, srcPort)));
    ports = put(ports, srcPort, inSet(lookup(ports, srcPort), True));
    ports = put(ports, direc, outSet(lookup(ports, direc), True));
}

← invocREv(_, this, fr, redirectPk, (pk, srcP))
← check if input and output channels are available

← block input and output channels

← invocEv(this, _, _, getPk, (pk, dirP))
The redirectPk method

Unit redirectPk(Packet pk, Direction srcPort){
    Direction direc = xFirstRouting(addressPk(pk), address);
    await (inState(lookup(ports,srcPort)) == True) &&
    (outState(lookup(ports,direc)) == True);

    ports = put(ports, srcPort,
               inSet(lookup(ports, srcPort), False));
    ports = put(ports, direc,
               outSet(lookup(ports, direc), False));

    Router r = rId(lookup(ports, direc));
    Fut<Unit> f = rlgetPk(pk, opposite(direc));
    await f?;

    ports = put(ports, srcPort,
               decreaseBuff(lookup(ports, srcPort)));
    ports = put(ports, srcPort,
               inSet(lookup(ports, srcPort), True));
    ports = put(ports, direc,
               outSet(lookup(ports, direc), True));
}

← invocREv(_, this, fr, redirectPk, (pk, srcP))
← check if input and output channels are available
← block input and output channels
← invocEv(this, _, _, getPk, (pk, dirP))
← release input and output channels
The redirectPk method

Unit redirectPk(Packet pk, Direction srcPort){
    Direction direc = xFirstRouting(addressPk(pk), address);
    await (inState(lookup(ports,srcPort)) == True) &&
    (outState(lookup(ports,direc)) == True);
    ports = put(ports, srcPort,
               inSet(lookup(ports, srcPort), False));
    ports = put(ports, direc,
               outSet(lookup(ports, direc), False));
    Router r = rId(lookup(ports, direc));
    Fut<Unit> f = r!getPk(pk, opposite(direc));
    await f?;
    ports = put(ports, srcPort,
               decreaseBuff(lookup(ports, srcPort)));
    ports = put(ports, srcPort,
               inSet(lookup(ports, srcPort), True));
    ports = put(ports, direc,
               outSet(lookup(ports, direc), True));
}

← invocREv(_, this, fr, redirectPk, (pk, srcP))
← check if input and output channels are available
← block input and output channels
← invocEv(this, _, _, getPk, (pk, dirP))
← release input and output channels
← compEv(this, fr, redirectPk, _)
Event sequences, indices and quantifiers

- The data structure of histories is sequence
- The consequences of this choice is the usage of quantifiers and indices
An invariant of the Router class: lockReleased

Every time a router terminates an execution of the redirectPk method, the input and output channels used to redirect the fetched packet are released, and the packet has been redirected to a neighbor router through an invocation of the getPk method.

\[
\forall fr \cdot \text{compEv}(this, fr, redirectPk, _) = \text{at}(h, \text{len}(h) - 1)
\]
\[
\Rightarrow
\]
\[
\exists i_1, i_2, pk, src, dir \cdot 0 < i_1 < i_2 < \text{len}(h) - 1 \land
\]
\[
(\text{invocREv}(_, this, fr, redirectPk, (pk, src)) = \text{at}(h, i_1) \land
\]
\[
\text{invocEv}(this, _, _, getPk, (pk, \text{opposite}(dir))) = \text{at}(h, i_2)) \land
\]
\[
(\text{inState}(\text{lookup}(ports, src)) \land \text{outState}(\text{lookup}(ports, dir)))
\]
An invariant of the Router class: `packetFetched`

Every time a router terminates an execution of the `getPk` method, it must either have sent an internal invocation to redirect the packet or have stored the packet in its `receivedPks` set.
Global Safety Properties

- $h$: local history
- $H$: global history
- $H/o$: projection from global history $H$ to object $o$
- $I_{\text{this}}(h)$: $\text{lockReleased} \land \text{packetFetched}$

A global history invariant can be obtained from the class invariants associated with all objects in the system, adding wellformedness of the global history.

$$I(H) \triangleq \text{wf}(H) \land \bigwedge_{(r: \text{RouterImp}) \in new_{ob}(H)} I_r(H/r)$$
Global Safety Properties

Every time a router $R$ terminates an execution of the redirectPk method, the pair of input and output channels used to redirect the fetched packet are released, and a neighbor router of $R$ must either have sent an internal invocation to redirect the packet further or have stored the packet in its $receivedPks$ set. Hence, the network does not drop any packets.

This implies:

- no packets are lost
The theorem prover KeY-ABS can be downloaded here: http://www.envisage-project.eu/?page_id=1558

Publications:
- C.C. Din and O. Owe, Compositional reasoning about active objects with shared futures. Formal Aspect of Computing. 2015
- C.C. Din, R. Bubel and R. Hähnle, KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS. CADE. 2015
Research Question 2

Class invariants should be readable and easy to reason about.

Is there a better way to formulate the class invariant without quantifying over the indices of the event sequences?
Trace-based class invariants

Work in progress:

- Idea 1: use explicit traces instead of finite sequences
- Idea 2: We need quantifiers for pattern matching but not for indices

Definition (The universal type quantifier \( \omega \))

The universal type quantifier \( \omega \) unwinds unbounded iteration. Let \( \Psi \) be a formula. \( \omega \ x \ . \ \Psi(x) \) captures any trace of the form \( \Psi(x_0) \ast \ast \Psi(x_1) \ast \ast \ldots \), where \( x_i \) range over the current model’s domain.
Every time a router terminates an execution of the redirectPk method, the input and output channels used to redirect the fetched packet are released, and the packet has been redirected to a neighbor router through an invocation of the getPk method.

\[ \omega_{fr, pk, src, dir}. \]

\[ ( \ldots \text{iRv}(this, _, fr, rPk, (pk, src)) \ll \text{iEv}(this, _, _, gPk, (pk, dir)) \ll [\text{portsReleased}]**\text{cEv}(this, fr, rPk, _) \]
Research Question 3

How to enable pre- and post-condition reasoning across asynchronous calls?
Disjoint-event semantics for ABS method calls

- $o$
- $o'$
- $o''$

- invocation
- invocation reaction
- completion
- completion reaction

Active Object Workshop, 28.09.2015, Nice, France | Crystal Chang Din (TUD) | 22
Disjoint-event semantics for ABS method calls

Pre-condition is not anymore guaranteed to hold upon method execution!
Summary

- We achieve scalable verification of ABS concurrent models by compositional reasoning about history-based observable behavior.
- The theorem prover KeY-ABS is available for downloaded at [http://www.envisage-project.eu/?page_id=1558](http://www.envisage-project.eu/?page_id=1558).
- Ongoing work:
  - Class invariants should be readable and easy to reason about.
  - Pre- and post-condition reasoning across asynchronous calls.