Asynchronous Method Contracts for ABS

Eduard Kamburjan
Crystal Chang Din
Einar Broch Johnsen
Reiner Hähnle
May 30, 2018
Method Contracts

Main Challenges

- \texttt{o!m()} – Decoupled call and start of execution
- \texttt{get} – Decoupled call and read of the return value
- \texttt{await} – Intermediate suspension points
Method Contracts

Main Challenges

- `o!m()` – Decoupled call and start of execution
- `get` – Decoupled call and read of the return value
- `await` – Intermediate suspension points

Core Ideas

- Annotate concurrency context
- Verify functional part with KeY
- Check context statically on composition
Specification: \text{o!m()} – Preconditions
Main Idea: Two preconditions

- Constraint on parameters (for caller) in interface
- Constraint on state (for previous process) in class

```java
interface I {
    /*@ requires i > 0; @*/
    Unit m(Int i);
}

class A(Rat r) implements I{
    /*@ requires r > 0; @*/
    Unit m(Int i){ ... }
}
```
Preconditions

Context preconditions

- Terminated methods which guarantee precondition
- Possibly run methods which preserve precondition

```java
interface I {
    /*@ requires i > 0; @*/
    Unit m(Int i);

class A(Rat r) implements I{
    /*@ requires r > 0;
    succeeds m2;
    overlaps m3;@*/
    Unit m(Int i){ ... }
```
Additional propagation step between specification and proof:

<table>
<thead>
<tr>
<th>Additional propagation step between specification and proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Add state-constraint to postcondition of all contracts of methods in <code>succeeds</code></td>
</tr>
<tr>
<td>- Add new spec. case to all methods in <code>overlaps with $\phi$</code> in pre- and postcondition</td>
</tr>
</tbody>
</table>
class A implements A{
  Rat r;
 /*@ requires i > 0; requires r > 0
  succeeds m;   overlaps up ... @*/
  Unit test(Int i){ ... }
 /*@ ensures sth @*/
  Unit up(){ r++; }
 /*@ ensures sth @*/
  Unit m(){ r = 10; }
}
class A implements A{
    Rat r;
   /*@ requires i > 0; requires r > 0
    succeeds m;    overlaps up ... @*/
    Unit test(Int i){ ... }
   /*@ requires r > 0 ensures r > 0 @*/
   /*@ ensures sth @@*/
    Unit up(){ r++; }
   /*@ ensures sth && r > 0 @*/
    Unit m(){ r = 10; }
}
Example

Check interleavings once main block is provided

```java
class A implements A{
  /*@ succeeds m;  overlaps none @*/
  Unit test(Int i){...}
  Unit m(){ ... }
  Unit m2(){ ... }
}
```

Not correct

```java
o!m();
2 o!test();
3 o!m2();
```

Not Correct

```java
1 await o!m();
2 o!test();
3 o!m2();
```

Correct

```java
1 await o!m();
2 await o!test();
3 o!m2();
```
Workflow

Contracts → Propagation → Enriched Contracts → KeY

Classes

MHP constraints → SACO

Main Block

Modular Concrete
Verification of Concurrency Constraints

- overlaps is May-Happen-in-Parallel/partial order reduction
- succeeds is MHP + dependency analysis

succeeds

- MHP gives a set of methods which will have terminated, if run before
- Dependency analysis on method starts
- More precision: Sorting with dependency analysis

Propagation degenerates to invariants!
Specification: get – Postconditions
Example

class A implements A{
    Rat r; Int c;
   /*@ ensures \result > 0 @*/
    Unit m(Fut<Int> f){
        Int i = f.get;
        return i;
    }
}

What knowledge do we have about i?
class A implements A{
    Rat r; Int c;

    /*@ ensures result > 0 @*/
    Int m2(){ return 10; }

    /*@ ensures result > 0 @*/
    Unit m(Fut<Int> f){
        /*@ readsFrom m2 @*/
        Int i = f.get;
        return i;
    }
}

- use points-to with main block to check annotations
- add condition during symbolic execution
Accessing the Postcondition

- No need to split postcondition:

\[
\text{ensures } \textbf{this}.i > 0 \land \textbf{this}.i < \text{result} \\
\exists j. j > 0 \land j < \text{result} \\
\exists j. \textbf{this}.i > 0 \land \textbf{this}.i < j
\]
Workflow

Contracts → Propagation

Enriched Contracts → KeY

MHP / p2 constraints

Classes

SACO

Main Block

Modular Concrete
Specification: `await` – Suspension
Example

```
1 class A implements A{
2   Rat r; Int c;
3   /*@ ensures r > c @*/
4   Unit m(){
5       r = c - 1;
6       await True;
7       r = c + 2;
8   }
9 }
```

Is a postcondition a condition for all suspension points?
Suspension Points

- In FormbaR we require more in some methods:
  - At the `await` we hold a lock, but not at the `return`
- Postcondition describes termination
- Suspension points get extra conditions

```plaintext
1 /*@ assume r < 0; */
2   ensures r < 0;
3   overlaps m3
4   succeeds m2
5 /* */
6   await c < 0;
```
Suspension Points

- Method names not fine-grained enough
- More control over interleavings needed

```javascript
Unit m(Fut<Unit> f, Fut<Unit> f2){
    s1;
    await f?;
    s2;
    await f2?;
    s3;
}
...
Suspension Points

- Mark beginning of CFG block
- Method name refers to last block

```java
Unit m(Fut<Unit> f, Fut<Unit> f2){
    s1;
    [atom: "bl1"] await f?;
    s2;
    await f2?;
    s3;
}
... 
/*@succeeds bl1;*/  await c < 0;
```

- Method contracts are special suspension contracts
```java
/*@ requires \varphi_m */
ensures \chi_m @*/

Unit m() {
  s1;
 /*@ requires \varphi_1 */
  ensures \chi_1 @*/
  [atom: "1"] await g;
  s2;
 /*@ requires \varphi_2 */
  ensures \chi_2 @*/
  [atom: "2"] await g;
  s3;
  return e;
}
```

\begin{align*}
\varphi_m & \quad \text{Unit } m() \{ \\
\chi_1 & \quad [\text{atom: } "1"] \text{ await } g \} \quad \text{atomic segment 1} \\
\varphi_1 & \quad s_2 \quad \} \quad \text{atomic segment 2} \\
\chi_2 & \quad [\text{atom: } "2"] \text{ await } g \} \quad \text{atomic segment 2} \\
\varphi_2 & \quad s_3 \quad \} \quad \text{atomic segment m} \\
\chi_m & \quad \text{return } e; \} \\
\end{align*}
Verification: Deduction and Composition
How to connect Contract and Analyses?

- Characterize contract in meta-trace logic
- Characterize analysis in meta-trace logic
- Connection Lemma: Success of analysis implies contract

- Deduction is special analysis
Encapsulation

Logical Characterization of Resolving Contract \( \text{resolve}(\text{resolve}_k, tr) \)

\[
\forall i \in \mathbb{N}. \ ev^{tr}[i] \models \text{futREv}(X, f, e, k) \rightarrow \\
\exists j \in \mathbb{N}. \ \bigvee_{m \in \text{resolve}_k} ev^{tr}[j] \models \text{futEv}(X', f, m, e)
\]
Encapsulation

Logical Characterization of Resolving Contract \( \text{resolve}(\text{resolve}_k, tr) \)

\[
\forall i \in \mathbb{N}. \ ev^{tr}[i] \models \text{futREv}(X, f, e, k) \rightarrow \\
\exists j \in \mathbb{N}. \ \bigvee_{m \in \text{resolve}_k} ev^{tr}[j] \models \text{futEv}(X', f, m, e)
\]

Logical Characterization of Points-To Analysis \( \text{points}(k, tr) \)

\[
\forall i \in \mathbb{N}. \ ev^{tr}[i] \models \text{futREv}(X, f, e, k) \rightarrow \\
\exists j \in \mathbb{N}. \ \bigvee_{m \in \text{p2}(k)} ev^{tr}[j] \models \text{futEv}(X', f, m, e)
\]
Encapsulation

Logical Characterization of Resolving Contract resolve(resolve_k, tr)

\[ \forall i \in \mathbb{N}. \ ev^{tr}[i] \models \text{futREv}(X, f, e, k) \rightarrow \]
\[ \exists j \in \mathbb{N}. \ \bigvee_{m \in \text{resolve}_k} ev^{tr}[j] \models \text{futEv}(X', f, m, e) \]

Logical Characterization of Points-To Analysis points(k, tr)

\[ \forall i \in \mathbb{N}. \ ev^{tr}[i] \models \text{futREv}(X, f, e, k) \rightarrow \]
\[ \exists j \in \mathbb{N}. \ \bigvee_{m \in \text{p2}(k)} ev^{tr}[j] \models \text{futEv}(X', f, m, e) \]

Connecting Lemma

\[ \forall tr. \ p2(k) \subseteq \text{resolve}_k \rightarrow (\text{points}(k, tr) \rightarrow \text{resolve} (\text{resolve}_k, tr)) \]
(get) \[ fresh(r, T), (\forall m \in \text{resolve}(k) \exists m(r)) \implies \{ v := r \} \{ T := T \cdot \text{futREv}(\text{this}, f, r, k) \}[s]\chi \]
\[ \implies [[\text{sync: } "k"] v = f.g\text{et}; s]\chi \]

(aw) \[ fresh(t, T) \implies U_A \{ T := T \cdot \text{suspEv}(\text{this}, F, M, i) \cdot t \cdot \text{reacEv}(\text{this}, F, M, i)(\phi_i \to [s]\chi) \}
\[ \implies [[\text{atom: } "i"] \text{ await } f?; s]\chi \]
Soundness

Coherence
A set of method contract is coherent after propagation
Soundness

Coherence
A set of method contract is coherent after propagation

Prgm-Soundness
Let Prgm be a program. A rule with premises $P_1 \ldots P_n$ and conclusion $C$ is Prgm-sound if for every $\beta$ and every partial trace $tr$ of Prgm the following holds: $\left( \wedge_{i \leq n} [P_i]_{tr, \beta} \right) \rightarrow [C]_{tr, \beta}$.

Rules depend on the program!

\[
\text{(get)} \quad fresh(r, T), \ (\bigvee_{m \in \text{resolve}(i)} \vec{m}(r)) \quad \Rightarrow \quad \{ v := r \} \{ T := T \cdot \text{futREv}(\text{this}, f, r, i) \}[s] \chi \\
\Rightarrow \quad \[[\text{sync: "i"}] \ v = f.\text{get}; s] \chi
\]
Prgm-Soundness of \((\text{get})\)

- One rule per synchronization point!
- Soundness of \((\text{get})\) is not compositional
  - Requires success of Points-To Analysis
  - Requires that all other method obligations are proven
  - Requires that all other \((\text{get})\)-rules are sound
- Proof that for every trace, every future read satisfies Prgm-soundness
- Proof per induction on the number of future read in trace
Induction Base: First synchronization

- Corresponding \((\text{get})\)-application is sound
- Requires that previous methods are proven, but these contain no future reads
Induction Base: First synchronization

- Corresponding (get)-application is sound
- Requires that previous methods are proven, but these contain no future reads

Induction Step: $n + 1$th synchronization

- Corresponding (get)-application is sound
- Requires that previous methods are proven, but by IH all (get)-application there were sound
Induction Base: First synchronization

- Corresponding (get)-application is sound
- Requires that previous methods are proven, but these contain no future reads

Induction Step: $n + 1$th synchronization

- Corresponding (get)-application is sound
- Requires that previous methods are proven, but by IH all (get)-application there were sound

Similar for (aw), obviously does not work for recursion
Global Soundness

Soundness of Compositional Reasoning

Let $M$ be a coherent set of method contracts. If

1. the PT, MHP and MHF analyses succeed on $M$

2. for each $M_m \in M$ the proof obligation can be shown, then the following holds for all $tr$ with Prgm $\downarrow$ $tr$:

$$\bigwedge_{M_m \in M} (\text{assert}(M_m, tr) \land \text{assume}(M_m, tr) \land \text{context}(M_m, tr) \land \text{resolve}(M_m, tr))$$
Conclusion
Contracts

Asynchronous method contracts

- Two preconditions
- One postcondition for termination
- One suspension contract per suspension point
- A set of methods guaranteeing the precondition
- A set of methods preserving the precondition

- Hides complexity in calculus: concurrency pushed out of KeY
- Complexity visible in specification (compared to JML)
**Better Rules (soon™)**

\[
\begin{align*}
\text{(get)} & \quad \frac{\text{fresh}(r), \bigvee_{m \in \text{resolve}(i)} \overline{m}(r)}{\Rightarrow \left[\text{sync: "i"} \right] v = f.\text{get};s} \chi \\
\text{(aw)} & \quad \frac{\Rightarrow \chi_i}{\Rightarrow \left[\text{atom: "i"} \right] \text{await } f?;s} \chi \\
\end{align*}
\]
Encapsulation of memory enforces encapsulation of specifications

- Cannot express global properties like protocols
- “The data I receive is a valid key for some internal map”
Future Work

- Implementation
- Method Contracts generated from Session Types
- Better Calculus
- Trace Logic/New Semantics
- Recursion
Summary

- Two preconditions: heap – parameters
- succeeds
- overlaps
- One postcondition: at termination
- Suspension contracts

Contracts → Propagation

Enriched Contracts → KeY

MHP / p2 constraints

SACO

Main Block

Thank you for your attention!