Asynchronous Method Contracts for ABS

Eduard Kamburjan

Crystal Chang Din

Einar Broch Johnsen

Reiner Hähnle

May 30, 2018

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

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: `o!m()` – Preconditions

Preconditions

Main Idea: Two preconditions

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

```
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

```
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){ ... }
```

Preconditions

Additional propagation step between specification and proof:

Additional propagation step between specification and proof

- Add state-constraint to postcondition of all contracts of methods in `succeeds`
- Add new spec. case to all methods in `overlaps` with ϕ in pre- and postcondition

Example

```
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; }
}
```

Example

```
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

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

Not correct

```
o!m();
o!test();
o!m2();
```

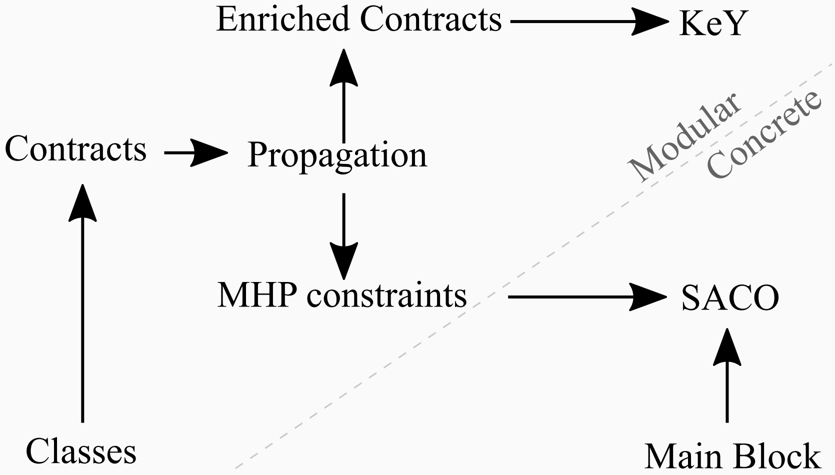
Not Correct

```
await o!m();
o!test();
o!m2();
```

Correct

```
await o!m();
await o!test();
o!m2();
```

Workflow


Enriched Contracts
KeY
Modular
Concrete
Contracts
Propagation
MHP constraints
SACO
Classes
Main Block

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`?

Accessing the Postcondition

```
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

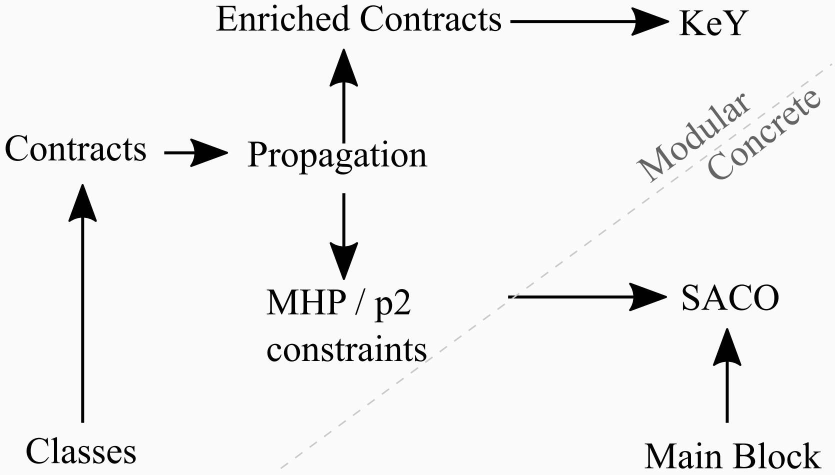
- No need to split postcondition:

```
ensures  this.i > 0 && this.i < result
```

$$\exists j. j > 0 \wedge j < \texttt{result}$$

$$\exists j. \textbf{this}.\texttt{i} > 0 \wedge \textbf{this}.\texttt{i} < j$$

Workflow


Enriched Contracts
KeY
Modular
Concrete
Contracts
Propagation
MHP / p2
constraints
SACO
Classes
Main Block

Specification: `await` – Suspension

Example

```
class A implements A{
  Rat r; Int c;
/*@ ensures r > c @*/
  Unit m(){
      r = c - 1;
      await True;
      r = c + 2;
  }
}
```

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

```
/*@ assume r < 0;
    ensures r < 0;
    overlaps m3
    succeeds m2
@*/
     await c < 0;
```

Suspension Points

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

```
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

```
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

Segment Structure

```
/*@ requires φm
      ensures χm @*/
Unit m() {
 s1;
 /*@ requires φ1
      ensures χ1 @*/
 [atom: "1"] await g;
 s2;
 /*@ requires φ2
      ensures χ2 @*/
 [atom: "2"] await g;
 s3;
 return e;
}
```

```
φm        Unit m() {
                s1                  } atomic
χ1   [atom: "1"] await g            } segment 1
φ1
                s2                  } atomic
χ2   [atom: "2"] await g            } segement 2
φ2
                s3                  } atomic
χm        return e; }               } segment m
```

Verification: Deduction and Composition

Encapsulation

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 $\mathsf{resolve}(\mathsf{resolve}_k, tr)$

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

Encapsulation

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

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

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

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

Encapsulation

Logical Characterization of Resolving Contract resolve(resolve$_k$, tr)

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

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

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

Connecting Lemma

$$\forall tr.\ \mathsf{p2}(k) \subseteq \mathsf{resolve}_k \rightarrow (\mathsf{points}(k, tr) \rightarrow \mathsf{resolve}(\mathsf{resolve}_k, tr))$$

Rules

$$\textbf{(get)}\ \frac{\mathit{fresh}(\mathtt{r}, \mathbb{T}),\ \left(\bigvee_{\mathtt{m} \in \mathsf{resolve}(k)} \widehat{\chi\mathtt{m}}(\mathtt{r})\right) \Longrightarrow \{\mathtt{v{:=}r}\}\{\mathbb{T} := \mathbb{T} \cdot \mathsf{futREv}(\mathsf{this}, \mathtt{f}, \mathtt{r}, k)\}[\mathtt{s}]\chi}{\Longrightarrow [\mathtt{[sync:\ "k"]\ v\ =\ f.}\mathbf{get}\mathtt{;s}]\chi}$$

$$\textbf{(aw)}\ \frac{\begin{array}{l}\Longrightarrow \{\mathbb{T} := \mathbb{T} \cdot \mathsf{suspEv}(\mathsf{this}, \mathbb{F}, \mathbb{M}, i)\}\chi_i \\ \mathit{fresh}(t, \mathbb{T}) \Longrightarrow U_A\{\mathbb{T} := \mathbb{T} \cdot \mathsf{suspEv}(\mathsf{this}, \mathbb{F}, \mathbb{M}, i) \cdot t \cdot \mathsf{reacEv}(\mathsf{this}, \mathbb{F}, \mathbb{M}, i)\}(\phi_i \rightarrow [\mathtt{s}]\chi)\end{array}}{\Longrightarrow [\mathtt{[atom:\ "i"]}\ \mathbf{await}\ \mathtt{f?;s}]\chi}$$

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 β and every partial trace tr of Prgm the following holds: $\left(\bigwedge_{i \leq n} [\![P_i]\!]_{tr,\beta}\right) \rightarrow [\![C]\!]_{tr,\beta}$.

Rules depend on the program!

$$\textbf{(get)}\ \frac{\mathit{fresh}(\mathtt{r}, \mathbb{T}),\ \left(\bigvee_{\mathtt{m} \in \mathsf{resolve}(i)} \widehat{\chi\mathtt{m}}(\mathtt{r})\right) \Longrightarrow \{\mathtt{v:=r}\}\{\mathbb{T} := \mathbb{T} \cdot \mathsf{futREv}(\mathsf{this}, \mathtt{f}, \mathtt{r}, i)\}[\mathtt{s}]\chi}{\Longrightarrow [\mathtt{[sync:\ "i"]\ v\ =\ f.}\textbf{get}\mathtt{;s}]\chi}$$

Prgm-**Soundness of (get)**

- One rule per synchronization point!
- Soundness of **(get)** is not compositional
  - Requires success of Points-To Analysis
  - Requires that all other method obligations are proven
  - Requires that all other **(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

Prgm-**Soundness of (get)**

Induction Base: First synchronization

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

Prgm-**Soundness of** **(get)**

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

Prgm-**Soundness of (get)**

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 $\mathcal{M}_{\mathtt{m}} \in M$ the proof obligation can be shown, then the following holds for all tr with $\text{Prgm} \Downarrow tr$:

$$\bigwedge_{\mathcal{M}_{\mathtt{m}} \in M} \left(\text{assert}(\mathcal{M}_{\mathtt{m}}, tr) \wedge \text{assume}(\mathcal{M}_{\mathtt{m}}, tr) \wedge \text{context}(\mathcal{M}_{\mathtt{m}}, tr) \wedge \text{resolve}(\mathcal{M}_{\mathtt{m}}, tr)\right)$$

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™)

$$\textbf{(get)}\ \frac{\textit{fresh}(\mathtt{r}),\ \left(\bigvee_{\mathtt{m}\in\mathsf{resolve}(i)} \widehat{\chi\mathtt{m}}(\mathtt{r})\right) \Longrightarrow \{\mathtt{v:=r}\}[\mathtt{s}]\chi}{\Longrightarrow [\mathtt{[sync:\ "i"]\ v\ =\ f.}\textbf{get}\mathtt{;s}]\chi}$$

$$\textbf{(aw)}\ \frac{\begin{array}{c}\Longrightarrow \chi_i \\ \Longrightarrow U_A(\phi_i \rightarrow [\mathtt{s}]\chi)\end{array}}{\Longrightarrow [\mathtt{[atom:\ "i"]}\ \textbf{await}\ \mathtt{f?;s}]\chi}$$

Limits

Encapsulation of memory enforces encapsulation of specifications

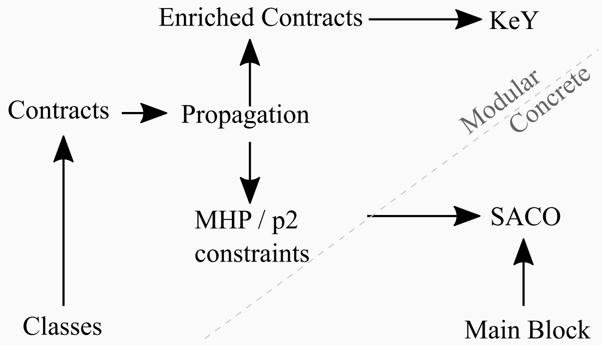
- Cannot express global properties like protocols
- “The data I receive is a valid key for some internal map”

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



Thank you for your attention!