Session-Based Compositional Verification on Actor-based Concurrent Systems

Session Types for ABS

Eduard Kamburjan, Crystal Chang Din, Tzu-Chun Chen

Motivation

- ABS language can be verified w.r.t. *methods preserving class invariants*
- Specification is written class-local directly in the verification logic

Motivation

TECHNISCHE UNIVERSITÄT DARMSTADT

- ABS language can be verified w.r.t. *methods preserving class invariants*
- Specification is written class-local directly in the verification logic

Aim

Use session types as specification language for global communication of ABS systems.

Motivation

TECHNISCHE UNIVERSITÄT DARMSTADT

- ABS language can be verified w.r.t. *methods preserving class invariants*
- Specification is written class-local directly in the verification logic

Aim

Use session types as specification language for
global communication of ABS systems.

- But session types are not defined for such a restrictive concurrency model

The ABS Concurrency Model

TECHNISCHE UNIVERSITÄT DARMSTADT

- ABS stands for Abstract Behavioural Specification
- Communication between objects are always **asynchronous method calls**
- No direct access to the fields of other objects
- Asynchronous method calls are realized through **futures**
- Futures can be passed around through method calls
- Object fields with future types
- At most one process is active on an object at a time
- Cooperative scheduling
  - scheduling points are made syntactically explicit in the code
  - **await** statement (on future or boolean expression)

The ABS Concurrency Model

- ABS stands for Abstract Behavioural Specification
- Communication between objects are always **asynchronous method calls**
- No direct access to the fields of other objects
- Asynchronous method calls are realized through **futures**
- ~~Futures can be passed around through method calls~~
- ~~Object fields with future types~~
- At most one process is active on an object at a time
- Cooperative scheduling
  - scheduling points are made syntactically explicit in the code
  - **await** statement (on future or ~~boolean expression~~)

An ABS Example (asynchronous call + future)

```
int n(int i){
  Fut<int> f = o!m(i);
  await f ?;
  int r = f.get;
  return r;
}
```

ABS-based Session Types

- Adapt session types to *futures*
  Instead of using arbitary *channels*
- Projections (global types to local types):
  global behavior → *object-local* behavior → *method-local* behavior
- Specify history-based class invariants based on the session types
- Verify the history-based class invariants using KeY-ABS theorem prover

Basic Projection from Global Types to Local Types

Global Types

$$::= \quad p \xrightarrow{f} q : m(S).G$$

Local Types

$$::= \quad q!_f m(S).L \mid p?_f m(S).L$$

Basic Projection from Global Types to Local Types

Global Types

$$::= \quad p \xrightarrow{f} q : m(S).G \mid q \downarrow f : (S).G$$

Local Types

$$::= \quad q!_f m(S).L \mid p?_f m(S).L \mid \text{Put } f : (S).L$$

Basic Projection from Global Types to Local Types

Global Types

$$::= \quad p \xrightarrow{f} q : m(S).G \mid q \downarrow f : (S).G \mid f \uparrow p : (S).G$$

Local Types

$$::= \quad q!_f m(S).L \mid p?_f m(S).L \mid \text{Put } f : (S).L \mid \text{Get } f : (S).L$$

Basic Projection from Global Types to Local Types

Global Types

$$::= \quad p \xrightarrow{f} q : m(S).G \mid q \downarrow f : (S).G \mid f \uparrow p : (S).G \mid \mathrm{Rel}(p, f).G \mid \mathrm{end}$$

Local Types

$$::= \quad q!_f m(S).L \mid p?_f m(S).L \mid \mathrm{Put}\ f : (S).L$$
$$\mid \mathrm{Get}\ f : (S).L \mid \mathrm{Aw}(f, f').L \mid \mathrm{React}\ f.L \mid \mathrm{end}$$

Basic Projection from Global Types to Local Types

Global Types

$$::= \quad p \xrightarrow{f} q : m(S).G \mid q \downarrow f : (S).G \mid f \uparrow p : (S).G \mid \mathrm{Rel}(p, f).G \mid \mathrm{end}$$

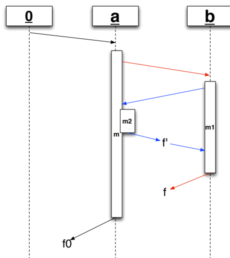
Local Types

$$::= \quad q!_f m(S).L \mid p?_f m(S).L \mid \mathrm{Put}\ f : (S).L$$
$$\mid \mathrm{Get}\ f : (S).L \mid \mathrm{Aw}(f, f').L \mid \mathrm{React}\ f.L \mid \mathrm{end}$$

- Global Rel only needed if multiple Aw could be possible
- React never needed in global type

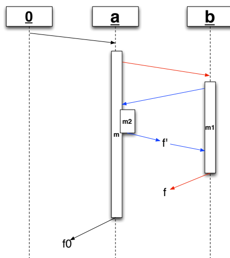
Example

$0 \xrightarrow{f_0} a : m. \quad a \xrightarrow{f} b : m_1. \quad b \xrightarrow{f'} a : m_2. \quad a \downarrow f'. \quad f' \uparrow b. \quad b \downarrow f. \quad a \downarrow f_0.$ **end**



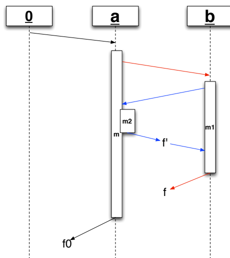
Example

	$0 \xrightarrow{f_0} a : m.$	$a \xrightarrow{f} b : m_1.$	$b \xrightarrow{f'} a : m_2.$	$a \downarrow f'.$	$f' \uparrow b.$	$b \downarrow f.$	$a \downarrow f_0.$	**end**
b		$?_f m_1.$	$!_{f'} m_2.$		Get f'.	Put f.		**end**
a	$?_{f_0} m.$	$!_f m_1.$	$\text{Aw}(f_0, f).?_{f'} m_2.$	Put f'.		React f_0.	Put f_0.	**end**



Example

	$0 \xrightarrow{f_0} a : m.$	$a \xrightarrow{f} b : m_1.$	$b \xrightarrow{f'} a : m_2.$	$a \downarrow f'.$	$f' \uparrow b.$	$b \downarrow f.$	$a \downarrow f_0.$	**end**
b		$?_f\, m_1.$	$!_{f'}\, m_2.$		Get f'.	Put f.		**end**
a	$?_{f_0}\, m.$	$!_f\, m_1.$	$\mathrm{Aw}(f_0, f).?_{f'}\, m_2.$	Put f'.		React f_0.	Put f_0.	**end**
m	$?_{f_0}\, m.$	$!_f\, m_1.$	$\mathrm{Aw}(f_0, f).$				Put f_0	
m_2			$?_{f'}\, m_2.$	Put f'				
m_1		$?_f\, m_1.$	$!_{f'}\, m_2.$		Get f'.	Put f		



Wellformed Global Types

- Consistency constraints on valid types, e.g. no reuse of futures
- For projecting on methods, keep track of
  - busy objects
  - currently computed futures
  - waiting futures
- Current status: relying on *unambiguous and scheduler-independent* control flow

Wellformed Global Types

TECHNISCHE UNIVERSITÄT DARMSTADT

- Consistency constraints on valid types, e.g. no reuse of futures
- For projecting on methods, keep track of
  - busy objects
  - currently computed futures
  - waiting futures
- Current status: relying on *unambiguous and scheduler-independent* control flow

$$0 \xrightarrow{f_0} a : m_0 \;.\; a \xrightarrow{f_1} b : m_1 \;.\; a \xrightarrow{f_2} c : m_2 \;.\; b \xrightarrow{f_3} c : m_2 \;.\; \ldots$$

Wellformed Global Types

TECHNISCHE UNIVERSITÄT DARMSTADT

- Consistency constraints on valid types, e.g. no reuse of futures
- For projecting on methods, keep track of
  - busy objects
  - currently computed futures
  - waiting futures
- Current status: relying on *unambiguous and scheduler-independent* control flow

$$0 \xrightarrow{f_0} a : m_0 \,.\, a \xrightarrow{f_1} b : m_1 \,.\, a \xrightarrow{f_2} c : m_2 \,.\, b \xrightarrow{f_3} c : m_2 \,.\, \ldots$$

Wellformed Global Types

TECHNISCHE UNIVERSITÄT DARMSTADT

- Consistency constraints on valid types, e.g. no reuse of futures
- For projecting on methods, keep track of
  - busy objects
  - currently computed futures
  - waiting futures
- Current status: relying on *unambiguous and scheduler-independent* control flow

$$0 \xrightarrow{f_0} a : m_0 \,.\, a \xrightarrow{f_1} b : m_1 \,.\, a \xrightarrow{f_2} c : m_2 \,.\, b \xrightarrow{f_3} c : m_2 \,.\, \ldots$$

$$0 \xrightarrow{f_0} a : m_0 \,.\, a \xrightarrow{f_1} b : m_1 \,.\, c \xrightarrow{f_2} e : m_2 \,.\, d \xrightarrow{f_3} f : m_2 \,.\, \ldots$$

Wellformed Global Types

TECHNISCHE UNIVERSITÄT DARMSTADT

- Consistency constraints on valid types, e.g. no reuse of futures
- For projecting on methods, keep track of
  - busy objects
  - currently computed futures
  - waiting futures
- Current status: relying on *unambiguous and scheduler-independent* control flow

$$0 \xrightarrow{f_0} a : m_0 \ . \ a \xrightarrow{f_1} b : m_1 \ . \ a \xrightarrow{f_2} c : m_2 \ . \ b \xrightarrow{f_3} c : m_2 \ . \ ...$$

$$0 \xrightarrow{f_0} a : m_0 \ . \ a \xrightarrow{f_1} b : m_1 \ . \ c \xrightarrow{f_2} e : m_2 \ . \ d \xrightarrow{f_3} f : m_2 \ . \ ...$$

Branching

TECHNISCHE UNIVERSITÄT DARMSTADT

Branching operator

$$p \begin{cases} l_1 & : G_1 \\ l_2 & : G_2 \\ ... & \\ l_n & : G_n \end{cases}$$

- With arbitary channels the choice is made by sending the label over a channel
- But futures cannot send arbitary data

Branching

TECHNISCHE UNIVERSITÄT DARMSTADT

Branching operator

$$p \begin{cases} l_1 & : p \xrightarrow{f_1} q_1 : m_1 \,.\, G'_1 \\ l_2 & : p \xrightarrow{f_2} q_2 : m_2 \,.\, G'_2 \\ \dots & \\ l_n & : p \xrightarrow{f_n} q_n : m_n \,.\, G'_n \end{cases}$$

- With arbitary channels the choice is made by sending the label over a channel
- But futures cannot send arbitary data
- Each choice must be made by calling a different method: $m_i \neq m_j$, if $q_i = q_j$
- Methodname takes role of branch-label

Repetition

TECHNISCHE UNIVERSITÄT DARMSTADT

Repetition operator

$$G^*$$

- Actions of futures are method calls and returns (and .get)
- If a method call is repeated, its return must also be repeated
- If a methods returns repeatedly, its call must also be repeated
- Futures used for repeated calls cannot be accessed afterwards

Repetition

Repetition operator

$$G^*$$

- Actions of futures are method calls and returns (and .get)
- If a method call is repeated, its return must also be repeated
- If a methods returns repeatedly, its call must also be repeated
- Futures used for repeated calls cannot be accessed afterwards

$$(p \xrightarrow{f} q : m)^* \,.\, q \downarrow f$$

Repetition

TECHNISCHE UNIVERSITÄT DARMSTADT

Repetition operator

$$G^*$$

- Actions of futures are method calls and returns (and .get)
- If a method call is repeated, its return must also be repeated
- If a methods returns repeatedly, its call must also be repeated
- Futures used for repeated calls cannot be accessed afterwards

$$(p \xrightarrow{f} q : m)^* \;.\; q \downarrow f$$

$$p \xrightarrow{f} q : m \;.\; (q \downarrow f)^*$$

Repetition

TECHNISCHE UNIVERSITÄT DARMSTADT

Repetition operator

$$G^*$$

- Actions of futures are method calls and returns (and .get)
- If a method call is repeated, its return must also be repeated
- If a methods returns repeatedly, its call must also be repeated
- Futures used for repeated calls cannot be accessed afterwards

$$(p \xrightarrow{f} q\!:\!m)^* \,.\, q \downarrow f$$

$$p \xrightarrow{f} q\!:\!m \,.\, (q \downarrow f)^*$$

$$(p \xrightarrow{f} q\!:\!m \,.\, q \downarrow f)^* \,.\, f \uparrow p$$

Composition

Conjuncture

If all methods satisfy their types in class A

Composition

TECHNISCHE UNIVERSITÄT DARMSTADT

Conjuncture

If all methods satisfy their types in class A and they are called according to the global type

Composition

Conjuncture

If all methods satisfy their types in class A and they are called according to the global type and no other communication is made

Composition

TECHNISCHE UNIVERSITÄT DARMSTADT

Conjuncture

If all methods satisfy their types in class A and they are called according to the global type and no other communication is made, then class A satisfies its local type.

Composition

Conjuncture

If all methods satisfy their types in class A and they are called according to the global type and no other communication is made, then class A satisfies its local type.

Conjuncture

If all classes satisfy their local types, and no other communication is made, then the whole system satisfies its global type. I.e. every communication is allowed by the session type.

TECHNISCHE UNIVERSITÄT DARMSTADT

Ongoing Work

Project is work in progress, missing parts:

- Correctness of composition
- Exploration of expressive power
- Wellformedness of repetition
- Scheduler configurations
- Extension to Protocol Types (Chen et Al.) with error handling
- Extension of KeY-ABS

Summary

TECHNISCHE UNIVERSITÄT DARMSTADT

- Using session types to specify ABS systems
- Adaption of session types to futures
- Projection to method-local types
- Verification with KeY-ABS

Thank you for your attention.