

Multiparty Session Types for Safe Runtime Adaptation in an Actor Language

Paul Harvey¹ Simon Fowler² Ornela Dardha³ Simon Gay³

¹Rakuten Institute of Technology, Japan

²School of Informatics, University of Edinburgh, UK

³School of Computing Science, University of Glasgow, UK

ABCD Meeting, December 2018

Overview

- **Adaptive** software is increasingly important for pervasive computing.
- Adaptation includes **discovering**, **replacing** and **communicating with** software components that are not part of the original system.
- **Ensemble** [Harvey 2015] is an actor-based language with support for adaptation.

Overview

- We designed and implemented **EnsembleS** by adding session types to Ensemble.
- **Static type checking** guarantees **safe runtime adaptation**.
- We extended the StMungo tool to **generate skeleton EnsembleS code** from Scribble local types.

EnsembleS language features

- Imperative actor-based language.
- Channels instead of mailboxes.
- Support for adaptation.

A simple EnsembleS program

```

1  type Isnd is
2      interface(out integer output)
3  type Ircv is
4      interface(in integer input)
5  stage home{
6      actor sender presents Isnd {
7          value = 1;
8          constructor() {}
9          behaviour {
10             send value on output;
11             value := value + 1;
12         } }
13
14     actor receiver presents Ircv {
15         constructor() {}
16         behaviour {
17             receive data from input;
18             printString("\nreceived:␣");
19             printInt(data);
20         } }
21     boot{
22         s = new sender();
23         r = new receiver();
24         establish topology(s, r);
25     } }

```

Session types in EnsembleS

- As well as **presenting** an interface, an actor can **follow** a session type.
- The session type is a **Scribble local type**.
- Typechecking checks the **sequence of messages** to and from other actors, and **connect/disconnect** actions.
- Individual messages are sent on **standard Ensemble channels**.

Buyer/Seller protocol in EnsembleS (1): global type

```
1  global protocol Bookstore (role Sell, role Buy1, role Buy2)
2  {
3    book(string) from Buy1 to Sell;
4    book(int) from Sell to Buy1;
5    quote(int) from Buy1 to Buy2;
6    choice at Buy2 {
7      agree(string) from Buy2 to Buy1, Sell;
8      transfer(int) from Buy1 to Sell;
9      transfer(int) from Buy2 to Sell;
10   } or {
11     quit(string)
12     from Buy2 to Buy1, Sell;
13   }
14 }
```

Buyer/Seller protocol in EnsembleS (2): local type

- Local types are generated by **projection**, as usual.

```
1 local protocol Buy1 (role Sell, self Buy1, role Buy2)
2 {
3   book(string) to Sell;
4   book(int) from Sell;
5   quote(int) to Buy2;
6   choice at Buy2 {
7     agree(string) from Buy2;
8     transfer(int) to Sell;
9   } or {
10    quit(string) from Buy2;
11  }
12 }
```


Buyer/Seller protocol in EnsembleS (3): actor interface

- The **interface** is generated from the local type: channels for each role and message type.

```
1  type Buy1_interface is interface(  
2    out {Seller, string} toSell_string,  
3    in {Seller, integer} fromSell_integer,  
4    out {Buy2, integer} toBuy2_integer,  
5    in {Buy2, Choice0} fromBuy2_agreequit,  
6    in {Buy2, string} fromBuy2_string,  
7    out {Sell, integer} toSell_integer,  
8  )
```

Buyer/Seller protocol in EnsembleS (4): actor definition

- Skeleton actor definitions are generated from the local types.
- Actors are also typechecked.

```

1  stage home {
2  actor Buy1A presents Buy1_interface
3          follows Buy1_session
4  {
5  constructor() {}
6  behaviour {
7  payload1 = "";
8  send payload1 on toSell_string;
9  receive payload2 from
10     fromSell_integer;
11  payload3 = 42;
12  send payload3 on
13     toBuy2_integer;
14  //Choice from other actor
15  receive payload4 from
16     fromBuy2_agreequit;
17
18  switch(payload4) {
19  case Choice0_agree:
20     receive payload5 from
21     fromBuy2_string;
22     payload6 = 42;
23     send payload6 on
24     toSell_integer;
25     break;
26  case Choice0_quit:
27     receive payload7 from
28     fromBuy2_string;
29     break;
30  }
31  }

```

Adaptation in Ensemble

- **Discover**: locate an actor with a given interface and satisfying a given query.
- **Install**: spawn a new actor instance at a specified stage.
- **Migrate**: move an executing actor to a different stage.
- **Replace**: replace an executing actor with a new actor instance with the same interface.
- **Interact**: connect to another actor and communicate with it.

Adaptation in EnsembleS, with session types

- **Discover**: locate an actor with a given interface and satisfying a given query **and a given session type**.
- **Replace**: replace an executing actor with a new actor instance with the same interface **and the same session type**.
- **Interact**: connect to another actor and communicate with it, **following its session type**.

EnsembleS discovery / replacement with session types (1)

```
1 actor fastA presents accountingI follows accountingSession
2 {
3   constructor() {}
4   behaviour {
5     receive data on input;
6     quicksort(data);
7     send data on output;
8   }
9 }
```

EnsembleS discovery / replacement with session types (2)

```
1 actor slowA presents accountingI
2   follows accountingSession {
3     pS = new property[2] of property("",0);
4     constructor() {
5       pS[0] := new property("serial",823);
6       pS[1] := new property("version",2);
7       publish pS;
8     }
9     behaviour {
10      receive data on input;
11      bubblesort(data);
12      send data on output;
13    }
14  }
15
16 query alpha() {
17   $serial==823 && $version<4;
18 }
```

EnsembleS discovery / replacement with session types (3)

```
1 actor master presents masterI{
2   constructor() { }
3   behaviour {
4     // find the slow actors with the query
5     actor_s =
6       findSessionActors(
7         accountingI,
8         accountingSession,
9         alpha());
10    // replace them with efficient versions
11    if(actor_s[0].length > 1){
12      replace actor_s[0]
13      with fastA();
14    }
15  }
16 }
```

Formalisation

- We have formalised a core calculus for EnsembleS.
- Operational semantics, type system, type preservation.
- A well-typed configuration proceeds until all of its actors have terminated, except for runtime situations such as attempting to use a disconnected channel, or absence of an actor matching a **discover** query.

Conclusion

- Session types for an existing, implemented (albeit experimental) actor language.
- Session types for adaptive features: discovery, dynamic connection, replacement.
- The formalisation matches the implementation; there are further possibilities for typechecking.