

Behavioural Prototypes

Roly Perera Simon J. Gay

School of Computing Science, University of Glasgow, UK
{roly.perera, simon.gay}@glasgow.ac.uk

Concurrent objects and multiparty compatibility. Data types describe values; behavioural types such as multiparty session types [4] and typestate [3] describe interactions. Here we introduce a simple actor language and show how *multiparty compatibility* [1] can be used to statically type-check systems of concurrent objects whose interfaces evolve dynamically in response to messages.

Our program in our language is a collection of communicating automata [2]. An asynchronous send of message $m(v)$ to p is written $p!m(v)$. A blocking receive from p is written $p?m(v)$, and binds the parameter v to the value sent. Our example models a simple software development workflow with four mutually recursive roles. The wavy underlining can be ignored for the moment.

In the code below, the `teamLead` starts the `devTeam` and then begins a `ReleaseCycle`. Once a release candidate at revision v is received from the `devTeam`, the `business` is informed. If the `business` accepts the release, it is tagged by the `teamLead` and the `devTeam` can stop. Alternatively the `business` can request another iteration, and the cycle repeats.

```
1 obj teamLead =
2   devTeam!start;
3   def ReleaseCycle =
4     devTeam?releaseCandidate(v: number):
5     business!releaseCandidate(v);
6     business?
7     iterate:
8       repository!tagRC(v);
9       devTeam!continue;
10      ReleaseCycle
11    accept:
12      repository!tagRelease(v);
13      devTeam!stop.
14  ReleaseCycle
```

The `repository` enforces a source control protocol for the organisation. Once a work unit has been committed by the `devTeam`, the `teamLead` must tag it appropriately before another work unit can be accepted. The `business` is a hard-coded test case that chooses non-deterministically between `iterate` and `accept` for the first release, and then accepts the second release without further ado.

```
1 obj repository =
2   def Connected =
3     devTeam?commit(v: number):
4     teamLead?
5     tagRC(v: number): Connected
6     tagRelease(v).
7   Connected
8
9 obj business =
10  teamLead?releaseCandidate(v: number):
11  teamLead!
12  iterate;
13  teamLead?releaseCandidate(v: number):
14  teamLead!accept.
15  accept.
```

The `devTeam` commits work units to the `repository`, simultaneously notifying the `teamLead`, and bumping the revision number at the end of each iteration. The interaction with the `math` library

shown here is asynchronous; a more realistic language would provide synchronous invocation as syntactic sugar.

```
1 obj devTeam =
2   teamLead?start:
3   def ReleaseCycle(v: number) =
4     // repository!commit(v);
5     teamLead!releaseCandidate(v);
6     teamLead?
7     continue:
8     math!plus(v, 1); math?val(w: number):
9     ReleaseCycle(w)
10    stop.
11  ReleaseCycle(0)
```

If the `commit` message to the `repository` is omitted, as shown, our implementation reports the compile-time errors shown as wavy lines. Red underlining indicates that there is a `!` state of the system which is stuck because that message cannot be delivered. Conversely, blue underlining indicates that there is a `?` state which is stuck because none of the permitted messages arrives. Technically, these errors indicate *multiparty incompatibility*: the objects cannot be safely composed because a coherent global session type cannot be derived which captures their interactions. (A global session type certifies that the only stuck states of a system are its terminal states [1].) However, note that we do not utilise a separate language of types: multiparty compatibility is checked directly for objects.

Behavioural prototyping. For modularity, it is important to be able to define robust subsystem boundaries. Our implementation does not support this yet, but in our “typeless” setting it would be natural for this role to be served by concrete, executable objects, rather than interfaces, and for the notion of implementation to be subsumed by behavioural subtyping. This would permit declarations such as the following, which refines the `business` object. It defines a specialised implementation of the `releaseCandidate` handler which asks a customer whether to `iterate` or `accept`:

```
1 obj business': business =
2   teamLead?releaseCandidate(v: number):
3   customer!evaluate(v);
4   customer?
5   reject(comments: string):
6   teamLead!iterate;
7   teamLead?releaseCandidate(v: number):
8   teamLead!accept.
9   ok: teamLead!accept.
```

Intuitively, this is a valid refinement of the `business` object because the customer interaction is private and preserves the observable behaviour of `business`. Implementation inheritance is mandatory.

Conclusion. We have shown some early features of our language and sketched an idea for incorporating subtyping. Several non-trivial challenges lie ahead. First, programs are not usually finite-state and so abstraction and finitisation techniques will be required for multiparty compatibility to remain decidable. Second, type errors reflect specific stuck states and so to diagnose them properly may require integrating the type system with a debugger. Finally, our language only supports systems with a fixed set of roles, and will need extending to support dynamically configured systems.

References

- [1] M. Carbone, F. Montesi, N. Yoshida, and C. Schurmann. Multiparty session types as coherence proofs. In *Concurrency Theory, 26th International Conference, CONCUR 2015*. Leibniz International Proceedings in Informatics, 2015.
- [2] P.-M. Deniélou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In F. V. Fomin, R. Freivalds, M. Kwiatkowska, and D. Peleg, editors, *Automata, Languages, and Programming*, volume 7966 of *LNCS*, pages 174–186. Springer Berlin Heidelberg, 2013.
- [3] R. Garcia, E. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, Oct. 2014.
- [4] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '08*, pages 273–284, New York, NY, USA, 2008. ACM.