Connecting Choreography Languages With Verified Stacks

Alejandro Gómez-Londoño[1] and Johannes Åman Pohjola[2]

[1] Chalmers University of Technology, Gothenburg, Sweden
alejandro.gomez@chalmers.se
[2] Data61/CSIRO, Sydney, Australia
johannes.amanpohjola@data61.csiro.au

Abstract

With ever increasing availability of verified stacks capable of guaranteeing end-to-end correctness on applications—like compilers (CakeML, CompCert) or even critical software systems (seL4)—one can now realistically write a program, along with a proof describing any desirable property, and have it compiled into a correct executable implementation of the original program. However, most of these approaches can only really deal with sequential programs and provide no support for reasoning about the correctness of multiple (concurrent) programs. To address these shortcomings, we propose a choreographic language where the behavior of a system consisting of several endpoints, is described on a global level, that can be subsequently projected and compiled into its individual components. We are developing an end-to-end proof of correctness that ensures i) the deadlock-freedom of the generated set of endpoints and ii) the preservation of all behavior of the system down to the binary level. Our implementation uses the verified CakeML compiler as a backend and takes advantages of its verified stack.

This extended abstract presents our ongoing work on connecting choreography languages with verified stacks. Our overarching goal is to develop an environment where programmers can write communicating systems as high-level protocol descriptions in the style of `Alice` → `Bob` notation, and then, easily generate executable code that is formally verified to correctly implement the protocol down to the machine-code level. To achieve our goal, we connect these two strands of work to provide both a convenient abstraction for representing such systems, and strong guarantees about their behaviour.

Choreographies present a global description of the interactions of a system in terms of the messages exchanged between its components [1] (endpoints). Its syntax resembles the description of a protocol, but it provides a more concrete definition of the system. For example, in lines 2 and 3 of Figure 1, `A` sends `B` the name of an `item` and `B` sends back its price by evaluating `price(x)`. This choreography, while simple, completely captures the interaction between `A` and `B`, abstracting away individual operations like `send` and `receive` as communications (→).

```
1  -- choreography
2  A[item]      → B.x
3  B[price(x)] → A.y
4
5  -- projection A
6  send(B,item)
7  receive(B,y)
8
9  -- projection B
10 receive(A,x)
11 send(B,price(x))
```

Figure 1: price-query choreography

Choreographies can be thought of as protocol descriptions, in the sense that they are descriptions that a set of programs—in the form of threads, processes, servers, etc—implement, and that as a whole, should exhibit the intended behaviour. Nevertheless, making sure a system complies with its specification is in general no easy task, and we have seen time and time again examples of protocols being implemented poorly [2]. What distinguishes choreographies is that they are programming languages, and hence provide concrete enough descriptions to generate implementations directly from them.

Endpoint projection (EPP) [6] generates an implementation for a choreography by translating each endpoint into a program, that when joined through parallel composition with all the others, should exhibit the same interactions as the original specification. This correctness property is know as the *endpoint projection theorem* (EPP theorem). Additionally, choreographies prevent any mismatch between `send` and `receive` operations by construction, since the language constructs for interaction describe the action of both parties. This in turn implies deadlock-freedom, which extends to any projected implementations thanks to the EPP theorem. This combination of features and guarantees is what makes choreographies a good candidate for describing communicating systems, and an interesting subject for our verification efforts.

Verified stacks aim to produce a set of tools and techniques—involving a combination of interactive theorem provers, SMT-solvers, and other tool-chains—that allow users to describe programs, and through a mechanized process, generate a representation in a target language (usually binary) while providing some formal guaranties about the behaviour of the resulting program w.r.t. the initial specification. Some examples of verified stacks are compilers like CakeML [4] and CompCert [5]. They provide a proof of correctness that guarantees that the observable behaviours of the generated executables are compatible with the behaviours of the source programs. Other approaches, like the seL4 microkernel [3], target specific programs and can include proofs of additional properties like security or deadlock-freedom.

By connecting a choreography language with a verified stack, one could provide end-to-end guarantees about communicating systems, using a simple and expressive interface with convenient properties. However, most EPP results in the literature target modelling languages like the π-calculus rather than concrete programming languages, and make no attempt to extend the results to executable representations of the protocol participants. Furthermore, there is limited support for representing (concurrent) interactions in most verified stacks, which restricts their usability to only sequential programs. We aim to address these issues by providing i) to the best of our knowledge, the first mechanized proof of the EPP theorem, ii) a proven correct choreographic compiler, and iii) a novel approach for dealing with open system specifications.

Our implementation uses the HOL4 theorem prover and is comprised of a choreographic language definition that gets projected using EPP to endpoints expressed in a process algebra which is similar to value-passing CCS, but also features explicit locations, internal and external choice, and a more concrete representation of data and local computations that is amenable to code generation. From there, a sequence of verified refinements of the endpoints gradually translates each process into concrete CakeML code. The first step eliminates the internal and external choice primitives by encoding them using send, receive and if-statements. The second step compiles from a process algebra where messages can have unlimited length to a process algebra where messages have a (configurable) maximum payload size by introducing a protocol which sends messages in smaller chunks. The (by now sequential) code at each location is translated into a functional program in the CakeML language. Finally, we can use the verified CakeML compiler [8] as a backend to compile the functional representation into concrete machine code for mainstream architectures (including x86-64 and ARMv8) such that the

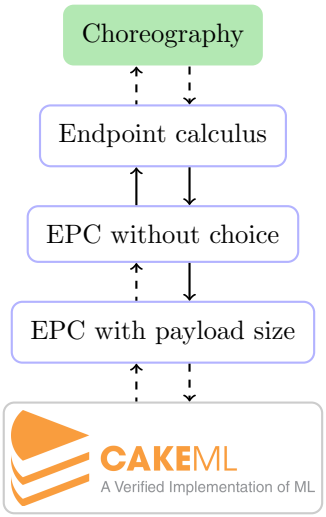

Figure 2: Roadmap

machine code preserves the observable behaviour of the CakeML program. By connecting the correctness proofs for each of these intermediate steps, we can obtain a top-level correctness statement that the communication behaviour and deadlock freedom properties of the choreography are preserved down to the machine code that runs it.

Local computations (e.g: `price(x)` in Figure 1) are shallowly embedded as functions in higher-order logic, hence can be directly translated by CakeML's proof-producing synthesis tool [7], or left underspecified to model external components. Finally, the underlying `send` and `receive` operations are implemented using CakeML's foreign function interface, which allows their use to be back-end agnostic (sockets, IPC, etc).

A high-level roadmap of our work is described in Figure 2, where each arrow signals a proof of semantic correspondence between the intermediate representations. Downwards arrows are semantic preservation proofs, and upward arrows are semantic reflection. Dashes account for sections pending verification. Additionally, proofs of confluence, deadlock-freedom, and structural congruence laws are in place for the semantics of our choreography language, and an extension for the CakeML FFI is being develop to allow for parameterized `send` and `receive` specifications.

In conclusion, by building on top of a flexible and robust verified stack like CakeML, and taking advantage of the strong guarantees and ease of use of choreographies, we believe this work will make the task of developing and maintaining verified systems a much more simple and cost-effective one.

References

[1] Laura Bocchi, Hernán Melgratti, and Emilio Tuosto. Resolving non-determinism in choreographies. In *European Symposium on Programming Languages and Systems*, pages 493–512. Springer, 2014.

[2] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488. ACM, 2014.

[3] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[4] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ml. In *ACM SIGPLAN Notices*, volume 49, pages 179–191. ACM, 2014.

[5] Xavier Leroy et al. The CompCert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[6] Fabrizio Montesi. Choreographic programming, 2014.

[7] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In *Proceedings of ICFP*, pages 115–126, 2012.

[8] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *Proceedings of ICFP*, pages 60–73, 2016.