Behavioural Types for Local-First Software (Artifact)

Roland Kuhn
Actyx AG, Kassel, Germany

Hernán Melgratti
University of Buenos Aires, Argentina
Conicet, Buenos Aires, Argentina

Emilio Tuosto
Gran Sasso Science Institute, L'Aquila, Italy

Abstract

This artifact supports the theory of *swarm protocols* presented in the related article. Specifically, following the top-down development typical of choreographic approaches, our artifact enables the specification of systems of peers communicating through an event notification mechanism from a *global* viewpoint which can then be projected to *local* specifications of peers, rendered as *machines*. To the best of our knowledge, ours is the first implementation of a behavioural type framework supporting the application of the principles of *local-first software* for network devices which collaborate on a common task while retaining full autonomy. The artifact can be integrated in the Actyx industrial platform; this proves this work a viable step towards reasoning about local-first and peer-to-peer software systems.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Distributed systems organizing principles; Software and its engineering → Distributed programming languages

Keywords and phrases Distributed coordination, local-first software, behavioural types, publish–subscribe, asynchronous communication

Digital Object Identifier 10.4230/DARTS.9.2.14

Funding Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233. Work partially funded by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems), by the PRO3 MUR project Software Quality, and by PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA – Advanced Space Technologies and Research Alliance. Work partly funded by the European Union (TaRDIS, 101093006).

Acknowledgements The authors also thank Daniela Marottoli for her help in the initial development of this artifact.

Related Article Roland Kuhn, Hernán Melgratti, and Emilio Tuosto, "Behavioural Types for Local-First Software", in 37th European Conference on Object-Oriented Programming (ECOOP 2023), LIPIcs, Vol. 263, pp. 15:1–15:28, 2023. `https://doi.org/10.4230/LIPIcs.ECOOP.2023.15`

Related Conference 37th European Conference on Object-Oriented Programming (ECOOP 2023), July 17–21, 2023, Seattle, Washington, United States

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2023 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

This artifact provides a prototype implementation of our type-checker and machine runner along with the code for our running example from the paper [1]. The tools can check and run this example. Summarising Section 3 of the paper and with reference to Fig. 2 (page 10 of our paper):

- `machine-check` verifies that events and their handlers are properly declared in TypeScript code, infers local types and subscriptions from TypeScript code, and finally relates events to states of swarm machines

- `machine-runner` uses TypeScript code as an API to execute the behaviour specified by local types
- `TypeChecking` checks for well-formedness of swarm protocols and subscriptions as defined in Section 6 of our paper, calculates projections, checks for equivalence of types inferred by machine-check and the corresponding projected ones. Moreover, the `rndMove` function described below performs simulation to explore possible admissible executions of our theoretical model.
- The artifact is applied to an example project to demonstrate the use of the inferred machine type to generically render a machine UI which allows the user to interact with machines by invoking their (enabled) commands.

Appendix A provides details on the usage of the artifact.

This artifact serves three main purposes: (*i*) To validate the results in our related ECOOP article, (*ii*) to demonstrate the feasibility and applicability of our type checking approach, and (*iii*) to enable the application of our theory in concrete applications.

2 Content

The artifact consists of four files:

- a Docker image for the `aarch64` processor architecture (e.g. Apple Silicon),
- a Docker image for the `amd64` processor architecture (e.g. most PCs),
- a README file in markdown format detailing how to run and explore the above images (also included as Appendix B), and
- an archive of the source code in case you want to make changes or use it in other ways.

We give more details on the source code structure in Appendix A.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at: `https://doi.org/10.5281/zenodo.7737188`.

4 Tested platforms

The artifact has been prepared as a Docker image for the `amd64` or `aarch64` (e.g. recent macbooks) processor architecture. Source code is provided as well.

5 License

The source code contained in the artifact (delineated by the file contents of `sources.tar.gz`) is available under license CC-BY-4.0. You are free to use the copy of Actyx and Actyx CLI contained within the Docker images in the context of exploring the artifact itself—further use requires a commercial license from Actyx AG. The remaining contents of the Docker images is available under their respective licenses.

6 MD5 sum of the artifact

4e62cef3573ecd16d61aab78c0b7d2ea

7 Size of the artifact

1.2GB

A Structure of the code

A.1 Directory /typechecking

- Formal syntax of `Machines` as finite equations, discussed in Section 5, (Equation 1 and below) corresponds to LType in `typechecking/src/LType.hs`.
- Formal syntax of swarm protocols as finite equations (Section 4), corresponds to `GType` in `typechecking/src/GType.hs`.
- Note that `LType` and `GType` correspond to particular cases of a general definition of infinite trees `ITree` represented as a set of equations, which are defined in `typechecking/src/ITree.hs`.
- Log operations (sublog relation, shuffling) in Section 2.4 are defined in `typechecking/src/Log.hs`
- Swarms and their operational semantics are in `typechecking/src/Swarm.hs`. It should be noted that there are two flavours of the semantics. Functions `local` and `receive` generate all possible continuations according to the rules [LOCAL] and [PROP]. In addition, we provide `rndMove` function which in case of several available moves in a swarm, it randomly selects the machine that randomly performs a [LOCAL] or [PROP] for a randomly selected command or log.
- Well-formedness properties of swarm protocols (Defs 4.1, 6.1, 6.3, 6.5, and 6.7) are implemented in `typechecking/src/Protocol.hs`.
- Projection operation in Def. 5.1 is given in `typechecking/src/Projection.hs`
- Effective type (Def. 7.3), Log equivalence (Def. 7.5) and eventually faithfullness (Def. 7.6) are given in `typechecking/src/Results.hs`
- The definition of the swarm protocol shown in the figure of Example 1.1 and then formalised in Example 4.2 is given in `typechecking/example/taxi-full/global.json`. Note that the definition is given as a state machine (as in the figure). The translations from finite-state machines to infinite trees and vice versa are implemented in `typechecking/src/Fsm.hs`. Their implementation is given through a type class, because it allows the transformation from finite-state machines (FSMs) to infinite trees of general types. We use such transformations both for swarm protocols and for the machines obtained by projections. Note that the projection operation in `typechecking/src/Projection.hs` generates infinite trees. To obtain their FSMs we transform trees to FSMs (i.e., for obtaining `P-projected-minimised.uml` when running `testcheck`).

See also `/typechecking/README.md` for more details.

A.2 Directory /machine-runner

- runtime library implementing the interpretation of asymmetric replicated state machines for use in real applications using the Actyx middleware
- definition of the `State` prototype in `src/types.ts`
- definition of the `@proto` decorator in `src/decorator.ts`
- definition of the `runMachine` function in `src/pond.ts`

A.3 Directory /machine-check

- build tool for inferring machine types from TypeScript code and using `/typechecking/typecheck` to verify them against a given swarm protocol
- type interence algorithm in `src/traverse.ts`

- conversion from runtime description of state machines to the JSON format understood by `typecheck` in `src/arsm.ts`
- conversion of TypeScript types into JSON schema in `src/typescript-json-schema.ts` (adapted from [2])

A.4 Directory /taxiRide

- machine definition in `src/machines.ts` (Listing 1 in the paper is a slightly reformatted excerpt from this file)
- illustration of adding a graphical UI in `src/ShowMachine.tsx` (using the React framework)

B Instructions

The artifact is delivered in the form of a docker image for the `amd64` or `aarch64` (e.g. recent macbooks) processor architecture. Please use `docker load -i docker_image.tar.gz` to load the image into your Docker daemon and note the name of its tag that Docker will print (should be `machines:latest`). You can inspect the contents and try out the build tools by starting a shell in the container:

```
docker run -it machines /bin/bash
```

The environment is a minimal debian, so you'll need to install your favourite editor if you want to make changes (e.g. `apt update && apt install emacs`). You start out in the `/taxiRide` directory, where the main tool demo is to run `npm run check`. This will analyse all machines in the project (see `src/machines.ts`), feed their inferred types to `../typechecking/typecheck`, and generate `src/proto.ts` to enable machine-runner to execute the machine.

You may try out introducing some errors to see how the check result changes. For example the protocol would not be well-formed if the taxi role ignored the `PassengerID` event (you can do this in `src/machines.ts:126` by removing the second argument to the `onSelected` method of `class AuctionP`). Another example could be to remove the emission of the `PassengerID` event in `execSelect` (line 119 in the same file), which will be flagged as the machine not matching its prescribed projection from the swarm protocol.

B.1 Playing with type-checking

From a docker shell go into folder `/typechecking/examples/taxi-full` and run

```
../../typeresults -g global.json -s subscription.json
```

to validate the conclusions in

- Example 3.2: Taxi example is log- and cmd-deterministic, hence it is deterministic
- Example 5.2: Taxi example is causal consistent wrt the given subscription
- Example 5.4: Taxi example is determinate wrt the given subscription
- Example 5.6: Taxi example is confusion-free wrt the given subscription
- Example 5.8: Taxi example is well-formed wrt the given subscription.

This command also produces the projections in Figure 3. More precisely, the command creates a file for each role in `typechecking/examples/taxi-full` yielding the minimised local types represented as `plantUml` state machines. For instance, `P-projected-minimised.uml` corresponds to the minimisation of the top machine in Fig. 3 (e.g., the two equivalent states in Fig. 3 that are sources of transitions labelled by `BidderId?` are represented by a unique state in the file `P-projected-minimised.uml`).

The `rndMove` function exploits projections to simulate our scenario in a swarm made of 3 replicas for each role. This function generates execution trases by randomly applying the semantic rules [LOCAL] and [PROP] (page 8) at each step. For each random, trace it checks that it reaches eventual consensus.

B.2 Live demo

Playing with the code is easier when you can use your usual code editor. To do that, download and unpack the `sources.tar.gz` and open a shell in the place where you unpacked it. The following command runs the image with your local sources of the `taxiRide` example mapped into the container so that your local edits will be picked up within the container:

```
docker run -d --name machines -p 1234:1234 -p 4454:4454 \
    -v ‘pwd‘/taxiRide/src:/taxiRide/src machines
```

This command starts the container as a background process (you can check it with `docker ps` or `docker logs machines`) and also initialises the Actyx runtime system, which is bundled in the container. Thereafter you can run commands inside this container, e.g. a shell with `docker exec -it machines /bin/bash` (note the `exec` instead of `run`). You can start the web browser demo with

```
docker exec machines npm start
```

Now you can take a look at a small demo app showing one passenger and two taxi machines with which you can interact by pointing your web browser to `http://localhost:1234`. If this fails or displays “Loading . . . ” make sure to include the `-p` options as shown in the `docker run` command above, and that your web browser is running on the same computer as docker.

With this setup, you can now edit the code in `taxiRide/src` and watch how that affects the app in the browser. Note that the swarm protocol checks will only be done again when restarting the `npm start` command given above.

References

1 Roland Kuhn, Hernán Melgratti, and Emilio Tuosto. Behavioural Types for Local-First Software. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:28, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2023.15.

2 YousefED. typescript-json-schema library, 2015-2023. URL: https://github.com/YousefED/typescript-json-schema/blob/master/typescript-json-schema.ts.