Liveness for Verification

Roly Perera Simon J. Gay
School of Computing Science, University of Glasgow, UK
{roly.perera, simon.gay}@glasgow.ac.uk

Abstract

We explore the use of liveness for interactive program verification for a simple concurrent object language. Our experimental IDE integrates two (formally dual) kinds of continuous testing into the development environment: *compatibility-checking*, which verifies an object's use of other objects, and *compliance-checking*, which verifies an object's claim to refine the behaviour of another object. Source code errors highlighted by the IDE are not static type errors but the reflection back to the source of runtime errors that occur in some execution of the system. We demonstrate our approach, and discuss opportunities and challenges.

1. Video submission

This accompanies the video submission at `https://vimeo.com/163716766`. We recommend reading the paper before watching the video.

2. Liveness for verification

Liveness [9, 12, 15] is often used to provide a tight feedback loop between program output and edits, reducing the cognitive burden on the programmer and supporting a more exploratory development style. In the present work, we use liveness to provide a tight feedback loop between *runtime errors* and edits, as a form of automated testing for concurrent programming that we call *language-integrated verification*. Our approach is related to *continuous testing* [11, 14], which runs tests automatically in the background and provides immediate feedback on test failures.

We explore this idea in a language based on *actors* [10], concurrent objects that in response to a message can explicitly transition to a new state offering different services. (From now on by the term "object" we mean concurrent object in the actor style.) The specific concurrency model is borrowed from communicating automata [5]: objects communicate asynchronously, maintain a separate FIFO mailbox for each client, and in every non-terminal state are either sending to or receiving from a unique other object. Further details on the language, programming model, and related work can be found in [13].

Language-integrated verification re-executes the program after an edit to revalidate its behaviour, rather than recompute its output. We explore the state space exhaustively, in the presence of non-determinism, performing two model-checking style analyses. *Compatibility-checking* verifies that objects can be safely composed, namely that every request for an interaction is eventually honoured. *Compliance-checking* verifies that when one system of objects is declared to refine the observable behaviour of another, every interaction supported by the refined system is supported by the refining system.

These analyses are based on *multiparty compatibility*, a notion from communicating automata and session types [3, 6, 7]. However, whereas multiparty compatibility is normally used to reason about *types*, here we apply it to objects. In our language there are no types, classes or interfaces; instead any concrete object or system of objects can serve as a specification of the behaviour of another. We describe our approach in § 3, with reference to the accompanying demo, and discuss limitations and future directions in § 4.

3. Overview of demo

As an example we model the interaction between a program committee and an author during a conference submission. On the left of Figure 1 overleaf, we define a system called `conf` with a single object `PC`; the blue underlining can be ignored for the moment. A *system* is simply one or more objects in parallel composition. The `author` object is left undefined; this is indicated by the name appearing in italics. The PC expects the author to submit a document, and then *non-deterministically chooses* between sending either `reject` or `conditionalAccept` back to the author. The direction of the triangle means either blocking receive (▸) or non-blocking send (◂); non-singleton choices are enclosed in braces `{...}`. A period denotes a terminal state.

Explicit non-determinism is a non-standard language feature which serves two important roles in our setting: it allows a single program or test case to capture multiple scenarios, and it allows a concrete object to be sufficiently abstract to serve as a specification. Whichever decision is made by the PC, a string of review comments is returned to the author. Here `string` denotes not a *type* but a *prototypical value* representing an unspecified string, consistent with our typeless approach. (A refinement of this system might choose to supply a concrete string instead.)

If the submission is accepted, the process enters an iterative phase: the author submits further revisions until the paper is either unconditionally accepted, or rejected. The iteration is implemented using a **`behaviour`** definition, which is simply a way of giving a name to a state. The state `Loop` is used twice here: recursively in the body of `revise`, and

```
system conf

obj PC
author▸submit(doc)
author◂{
   reject(string).
   conditionalAccept(string)
      behaviour Loop
         author▸submit(doc)
         author◂{
            reject(string).
            revise(string)
               Loop
            accept
               author◂artifactReq
               author▸{
                  decline.
                  provide(URL).
               }
         }
      Loop
}
```

```
system conf': conf

obj PC
author▸submit(doc)
author◂{
   accept.
   reject(string).
   conditionalAccept(string)
      behaviour Loop
         author▸{
            submit(doc)
               author◂{
                  reject(string).
                  revise(string)
                     Loop
                  accept
                     author◂artifactReq
                     author▸{
                        provide(URL)
                           artifact◂{
                              certify.
                              noCertify.
                           }
                     }
               }
         }
      Loop
}
```

Figure 1. Live compliance-checking

also immediately after the definition of `Loop`, as the body of `conditionalAccept`. If the paper is eventually accepted, an artifact request is issued, which the author may decline, or respond to by providing a URL pointing to the artifact.

We now discuss our two automated verification features: compliance-checking (§ 3.1) and compatibility-checking (§ 3.2). These involve executing all possible paths of the system and verifying that every reachable configuration is good. Compliance-checking is formally dual to compatibility-checking: to comply with an object (*qua* behavioural specification) is to be compatible with its dual, where one dualises an object by turning sends into receives and vice versa.

3.1 Compliance-checking

Figure 1 illustrates compliance-checking, where we verify that one system has the observable behaviour of another. On the right, the programmer defines a new system `conf'` which uses the colon syntax shown to declare that it implements `conf`. A number of compliance errors are detected in various states and reflected back to the relevant part of the source code. A convention we adopt for visualising errors is that they are shown from the vantage point of the system which has the focus, in this case `conf'`.

Thus, the blue underlining on `decline` that we disregarded earlier reflects a state in `conf` where the PC accepts a `decline` message from the author which the corresponding state in `conf'` does not support. The underlining of `decline` in `conf` should be understood as a convenient way of indicating its *absence* from `conf'`; other approaches are certainly possible. Dually, the red underlining of `accept` in `conf'` reflects a state where the PC sends an `accept` message to the author that the corresponding state in `conf` does not permit.

Finally, the red underlining on the name `artifact` reflects a state requiring an interaction with the object `artifact`, whereas the corresponding state in `conf` is terminal, as indicated by the period following `provide(URL)`.

3.2 Compatibility-checking

For compatibility-checking, we verify that the objects in a system compose in a safe way. In this example the programmer is able to build the `author` object interactively, using the compatibility errors to guide the implementation. As part of the implementation, we introduce another object, `coauthor` (left undefined), which the author consults in order to decide how to proceed if the paper is conditionally accepted.

Figure 2 shows an interim implementation, with errors which are again relativised to the system with focus, in this case `author`. The red wavey underlining on `reject` on the left reflects a state in which the author can only handle `revise` or `accept`, but the PC wants to `reject`. The blue underlining on

```
system conf

obj PC
author▸submit(doc)
author◂{
   reject(string).
   conditionalAccept(string)
      behaviour Loop
         author▸submit(doc)
         author◂{
            reject(string).
            revise(string)
               Loop
            accept
               author◂artifactReq
               author▸{
                  decline.
                  provide(URL).
               }
         }
      Loop
}
```

```
system author
using conf

obj author
PC◂submit("my paper")
PC▸{
   reject(str)
      coauthor◂rejected.
   conditionalAccept(str)
      behaviour Revise
         PC◂{
            submit(string)
               PC▸{
                  revise(str)
                     Revise
                  accept
                     PC▸artifactReq
                     PC◂provide("http://myurl.com").
            }
         }
      coauthor◂consult(str)
      coauthor▸{
         continue
            Revise
         withdraw
            PC◂withdraw.
     }
}
```

Figure 2. Live compatibility-checking

`revise` and `accept` in `author` reflect the same runtime error, and is in effect complementary to the red underlining on `reject` in `PC`.

The red error on `withdraw` on the right and the blue error on `submit` can be understood in the same mutually complementary way, but with the polarity reversed: the author is trying to send a `withdraw` message to the PC in a state where the PC will only accept `submit`. At present the UI does not make the connection between complementary errors apparent.

4. Conclusions and challenges

We described a prototypical IDE where errors reported to the user are not *type errors* but *runtime errors* that occurs in some reachable configuration of the system. The programmer works in the context of an active *system*, which is simply a set of objects composed in parallel; some represent application components being developed or tested, and others serve as "mock objects" or test cases representing exemplar scenarios. The programmer is responsible for defining each system to be small enough for exhaustive checking yet representative enough to give her confidence that the application feature it validates is correct.

In return, our implementation performs exhaustive checking automatically and provides a formal guarantee that execution paths validated in the IDE will execute correctly under an asynchronous semantics based on message queues "in the wild". Moreover any execution path which is valid for the system remains valid if an object is replaced by a compliant refinement of that object. Formalising the metatheory corresponding to these guarantees is work-in-progress.

The current implementation is naive: there are significant limitations relating to the language (§ 4.1), verification methods (§ 4.2), programming model (§ 4.3) and scalability (§ 4.4) which we intend to address in the future.

4.1 Language features

Our language lacks local and dynamically allocated objects, making it only suitable for toy examples. The formalism of communicating automata is extended with dynamic allocation in [4]; we plan to adapt multiparty compatibility to this setting. Another language feature we consider essential is inheritance, which requires a coinductive definition for communicating automata. Our compliance-testing is analogous to Java `implements`, rather than Java `extends`.

4.2 Verification methods

Like testing in general, but in contrast to a type system, our verification method is complete rather than sound: it potentially generates false positives rather than false negatives. One possible route to increased coverage, whilst staying faithful to our concrete, execution-oriented approach, is symbolic execution: this would allow individual tests to cover multiple executions, and (soundly) reduce the number of states that require explicit checking. Symbolic execution may also be needed to verify programs with free variables, a situation which arises often in our approach but which we have not properly considered yet.

4.3 User interface and programming model

Our prototypical IDE is based on a conventional text editor, with execution errors projected onto the source code. This presents a familiar user interface but one unsuited to the actual task, which is understanding and debugging problematic configurations (execution states). In future work, we plan to integrate a debugger with the editor, so that clicking on an error jumps to the corresponding problematic configuration, allowing the programmer to see what went wrong.

We would also like to combine our use of liveness for error reporting with liveness for visualising output. One idea would be to introduce a primitive object into our language representing a console or drawing canvas, and then treat the sequence of messages sent to that object as the program's output. Since we already explore every possible execution path for verification purposes, computing all possible outputs would incur no additional cost.

4.4 Efficient implementation

Modern software development workflows, such as test-driven development [2], are "incremental", in that they emphasise verifying *changes* to the program, rather than the whole program. We plan to exploit this by applying techniques from *incremental computation* [1, 8] to our compatibility-testing and compliance-testing algorithms. For certain kinds of edit, we should be able to incrementally update the analysis rather than recompute it from scratch. This is probably essential if our analyses are to scale to non-toy examples whilst remaining responsive enough for interactive use.

References

[1] U. A. Acar. *Self-Adjusting Computation.* PhD thesis, Department of Computing Science, Carnegie Mellon University, 2005.

[2] K. Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Company, Inc., Boston, MA, USA, 2002.

[3] L. Bocchi, J. Lange, and N. Yoshida. Meeting Deadlines Together. In L. Aceto and D. de Frutos Escrig, editors, *Concurrency Theory, 26th International Conference, CONCUR 2015*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 283–296, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] B. Bollig, A. Cyriac, L. Hélouët, A. Kara, and T. Schwentick. Dynamic communicating automata and branching high-level MSCs. In A.-H. Dediu, C. Martín-Vide, and B. Truthe, editors, *Language and Automata Theory and Applications*, volume 7810 of *Lecture Notes in Computer Science*, pages 177–189. Springer Berlin Heidelberg, 2013.

[5] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, Apr. 1983.

[6] 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.

[7] 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.

[8] M. A. Hammer, J. Dunfield, K. Headley, N. Labich, J. S. Foster, M. Hicks, and D. Van Horn. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 748–766, New York, NY, USA, 2015. ACM.

[9] C. M. Hancock. *Real-time programming and the big ideas of computational literacy.* PhD thesis, Massachusetts Institute of Technology, 2003.

[10] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[11] L. Madeyski and M. Kawalerowicz. Continuous test-driven development - A novel agile software development practice and supporting tool. In *ENASE 2013 - Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 260–267, 2013.

[12] S. McDirmid. Living it up with a live programming language. In *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 623–638, New York, NY, USA, 2007. ACM.

[13] R. Perera, J. Lange, and S. Gay. Multiparty compatibility for concurrent objects. In *9th Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software (PLACES)*, EPTCS. Open Publishing Association, 2016.

[14] D. Saff and M. D. Ernst. Continuous testing in Eclipse. In *2nd Eclipse Technology Exchange Workshop (eTX)*, Barcelona, Spain, 2004.

[15] S. L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing*, 1(2), 1990.