Correct Message-Passing Software Systems

Current and Future Research Challenges

Jorge A. Pérez
Bernoulli Institute for Mathematics, Computer Science, and Artificial Intellgence
University of Groningen
The Netherlands
j.a.perez@rug.nl

Abstract

Many critical software systems rely on message-passing between distributed artefacts, such as (micro)services. To a large extent, certifying system correctness amounts to ensuring that these artefacts respect some intended protocol.

Rooted in programming models from concurrency theory, *behavioral type systems* are a rigorous verification technique for message-passing programs. While these type systems are well understood by now, a main challenge remains: as their precision varies ostensibly, they induce seemingly unrelated forms of correctness. Since behavioral type systems still lack unifying foundations, it is hard to establish a most necessary dialogue with other researchers and software practitioners.

This note briefly introduces behavioral type systems and describes current research that targets the above challenge, framed within my VIDI career grant, recently awarded. It also suggests links between type-based program verification and other approaches in the intersection of formal methods, programming languages, and software engineering.

Keywords message-passing concurrency, verification, types

1 Communication Correctness

Ensuring software systems correct is widely recognized as a societal challenge (see, e.g., [?]). Establishing correctness for concurrent programs is much harder than for sequential programs. For message-passing programs, we may define *communication correctness* as the interplay of four properties:

- *Fidelity*: programs respect interaction protocols;
- *Safety*: programs do not get into errors, e.g., communication mismatches;
- *Deadlock-freedom*: programs do not get stuck;
- *Termination*: programs do not have infinite internal runs.

Certifying communication correctness is notoriously hard. Developers need *effective* and *practical* verification tools. Effective tools detect insidious bugs (e.g., communication errors, protocol mismatches, deadlocked services) before programs are run. Practical tools reconcile the *programming view* that developers understand well (where programs are implemented) and the *engineering view* needed to build large systems (where protocols are conceived).

PL'17, January 01–03, 2017, New York, NY, USA
2017.

2 Behavioral Types

Framed within concurrency theory and programming languages, *behavioral type systems* (or just *behavioral types* [?]) are a rigorous technique for certifying that message-passing programs satisfy communication correctness.

From Data Types to Behavioral Types Type systems prevent the occurrence of errors during the execution of a program [?]. The most basic property of a type system, *soundness*, ensures that "well-typed programs can't go wrong" [?].

While usual data types classify values, behavioral types organize concurrent interactions into *communication structures*, which are explicit for developers and architects [?]. Implementation agnostic, behavioral types are *effective*: the interaction sequences they codify are useful to exclude bugs that jeopardize communication correctness. Behavioral types are *practical*: communication structures can be compositionally expressed at different abstraction levels. Soundness of a behavioral type system typically ensures fidelity and safety; deadlock-freedom and termination are harder to enforce.

Foundations and (Too Many) Variants Behavioral types are usually defined on top of *process calculi*, formal languages that treat concurrent processes much like the λ-calculus treats computable functions. The π-calculus [?], the paradigmatic calculus of concurrency, is the specification language for many behavioral type systems. The intent is to define robust type systems for the π-calculus, and then transfer such foundations to specific languages. This is how verification tools based on behavioral types have been successfully developed for Haskell [?], Go [? ?], and Scala [?].

Many behavioral type systems exist; see [?] for a survey. Primarily intended as *static* verification techniques, which enforce correctness prior to a program's execution, behavioral types may help also in defining *dynamic* verification techniques that couple message-passing programs with so-called *monitors* that ensure fidelity and safety at run-time.

3 Two Pressing Challenges

While specific theories of behavioral types (most notably, *session types* [? ?]) are gradually reaching maturity, the body of knowledge on behavioral type systems as a whole is still immature, as two basic issues still lack proper justifications.

First, different behavioral type systems enforce *different notions of communication correctness*: hence, a program accepted as well-typed (i.e., communication-correct) by one type system could be ruled out as incorrect by another.

Second, and related to the above point, since the precise relations between different behavioral types are still poorly understood, their derived verification tools are *hardly interoperable*. As such, we still do not know how to combine distinct verification tools in the formal analysis of communication correctness in complex software systems.

Our ongoing research, recently supported by a VIDI career grant, aims at jointly tackling both challenges.[1]

In short, my VIDI career grant aims to unify distinct notions of communication correctness induced by different theories of behavioral types. To this end, various verification techniques for message-passing programs will be rigorously related in terms of their *relative expressiveness* [**?**]. To articulate a unified theory of correctness, we will crucially rely on the Curry-Howard correspondence for concurrency [**?**], the most principled link between concurrency and logic. Our preliminary results [**? ? ?**] are rather promising. We plan to validate these foundational results through case studies and tool prototypes.

4 A Research Agenda for the Future?

Tackling the two above challenges appears to be a pre-requisite step for addressing other relevant research questions in the intersection of formal methods, programming languages, and software engineering. Such questions include:

- ***Integration of techniques***. Can we combine techniques based on behavioral types with complementary techniques developed by (Dutch) researchers on formal methods, such as model checking and (concurrent) program logics?
- ***Cyber-security and trustworthiness***. Verification based on behavioral types uses protocols as key abstraction for enforcing communication correctness. There is potential for using this abstraction in other settings. For instance, can we reconcile this view of protocols with the one typically required to certify *security protocols* in critical software systems? (We have given a preliminary answer in [**?**].)
- ***Industrial transfer and impact***. To what extent verification based on behavioral types has a place in validation phases used by (Dutch) software industries?
- ***Usability***. Are verification frameworks based on behavioral types usable by software developers in practice? Can practice inform further theoretical research on type-based program verification? (A preliminary answer is in [**?**].)

While this list is by no means exhaustive, in our opinion these questions already suggest potential grounds for collaboration with other (Dutch) researchers, as well as also medium- and long-term challenges to be tackled in cooperation with practitioners and software industries.

[1]See http://www.jperez.nl/vidi for details.