LaDeDa: Languages for Debuggable Distributed Algorithms

Mark S. Miller
Google, Inc.

Tom Van Cutsem*
Vrije Universiteit Brussel

1 Language as Notation for Incorrect Programs

When programming language designs are presented, the examples are almost exclusively of correct programs. Most attention of programming language designers is indeed on the beauty and elegance of correct programs. For incorrect programs, great design attention is paid to catching errors early—such as fancy static type systems—so that many incorrect programs are never run.

Due to the success of these efforts, many programs are either correct or inadmissible, conserving on the need for programmer attention. As a result, most of the attention working programmers spend looking at code is spent debugging incorrect running code. Often this is code written by others and only partially understood. What properties should such code have? How can programming language design encourage incorrect programs to have those properties that facilitate debugging?

Distributed programs introduce additional difficult bugs of a different character. How should distributed language design facilitate the debugging of distributed programs?

We explain how these considerations have affected four distributed language designs (E [1], AmbientTalk [5], Joe-E/Waterken [4], Dr. SES [2]) and one distributed debugging tool (Causeway [3]).

2 Bounding Boxes for Answering Questions

When debugging, you're doing detective work. You do not need to understand the program as a whole, and often you cannot afford to. Rather, you're trying to track down a particular anomaly: Why did *this* bad thing happen? How much of the program is relevant? How much of its execution trace?

Ray tracing algorithms raise an analogous question: Of all the complex shapes in the scene, which of them intersect the ray? Their elegant solution is a system of *simple* bounding boxes so most of the scene can be cheaply

*Tom Van Cutsem is a post-doctoral Fellow of the Research Foundation, Flanders (FWO) disqualified, so that we can afford the complex calculations needed for the rest. Likewise, we need to disqualify most of the program from relevance to answering questions relevant to debugging. Possible causal influence is the most debugging-relevant question. The notations in which we express programs form the data structure we are searching.

- Mostly-functional programming bounds our worries about side effects to those parts of the program that need side effects for their expression.
- Strict lexical scoping combined with call-by-value argument passing bounds our worries about what code may have assigned to a given location.
- Encapsulation bounds our worries about what code may have directly violated a local invariant.
- Object-capability rules and style—Defensive Consistency and the Principle of Least Authority—bound worries about indirect invariant violations.
- Conventional sequential control flow bounds our worries about plan interference to those intervals when invariants are suspended.
- Pure message-passing concurrency bounds our worries about possible non-sequential interleavings to arrival order non-determinacy.
- Monotonic order-independent state transitions further bound indeterminacy. (Example: single-assignment of promises or logic variables.)
- Pure communicating event-loop concurrency, by avoiding blocking receives, bounds our worries about distributed invariants to non-stack state.
- Broken promise contagion bounds asynchronous failure handling to data dependencies.
- Causality tracing bounds worries about prior corruption to *happened before*.
- In a sequential debugger, visually emphasizing stack order over process order helps direct our suspicions to the more likely suspects first.
- When visualizing causality traces, emphasizing message-order over process order helps direct our suspicions to the more likely suspects first.

3 Case study: message passing

E—a pure event-loop-concurrent distributed object-capability language—has two message passing constructs, the *immediate call* (written ".") and the *eventual send* (written "←"). Each provides strong side effect guarantees, with opposite strengths and weaknesses. The familiar "b.foo(c)" immediately transfers control to b, which is necessarily local, suspending the caller until b returns. By contrast "bP ← foo(c)" queues, in the event loop hosting b, the need to deliver the foo message to b. bP denotes a promise to b, indicating that b may be remote. Whether or not b is remote, this delivery only happens in a separate turn of the event loop, starting from an empty stack.

Table 1 summarizes the advantages and disadvantages of these two message passing constructs. Both provide a strong set of complementary guarantees.

a performs:	**Immediate call** `b.foo(c)`	**Eventual send** `b<-foo(c)`
Virtue	No interleaving occurs between `a` calling `foo` and `foo` being called on `b`.	`a` proceeds and can safely repair suspended invariants before `foo` can ever affect its heap. Likewise, `b` starts processing `foo` from an empty stack, so there is no need to consider suspended invariants on the stack. `b` can assume all invariants have already been restored.
Hazard	`b` gets control while `a` is suspended, introducing potential plan interference if `b` violates `a`'s invariants.	Arbitrary code may have run between `a` sending `foo` and `b` executing `foo`. Therefore `b` must recheck all stateful assumptions on entry, other than restored invariants.

Table 1: Immediate call versus Eventual send

4 Summary

This position paper makes the case for *debuggable* distributed programming languages. Based on our prior experience in building distributed languages and debugging tools, we put forward a number of language properties that aid the programmer in reasoning about possibly faulty (distributed) code.

References

[1] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229, April 2005.

[2] M. S. Miller. Dr. ses: Distributed resilient secure ecmascript, April 2010. `http://es-lab.googlecode.com/files/dr-ses.pdf`.

[3] T. Stanley, T. Close, and M. S. Miller. Causeway: A message-oriented distributed debugger. Technical Report HPL-2009-78, HP Labs, April 2009.

[4] M. Stiegler and J. Tie. Introduction to waterken programming. Technical Report HPL-2010-89, HP Labs, August 2010.

[5] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Inter. Conf. of the Chilean Computer Science Society (SCCC)*, pages 3–12. IEEE Computer Society, 2007.