Applied Semantics: Specifying and Developing Abstractions for Distributed Computation (Grand Challenge Discussion Paper – GC2, GC4, and GC6)

Peter Sewell Keith Wansbrough

Computer Laboratory, University of Cambridge
http://www.cl.cam.ac.uk/users/{pes20,kw217}
February 2, 2004

Introduction

Over the last six years or so much of our research, together with colleagues, has aimed at establishing solid foundations for large-scale distributed computation: applying semantic techniques (existing or newly-developed) to real-world system problems. In this paper we briefly reflect on this work, discussing its relationship to the Grand Challenge proposals

GC2: Science for Global Ubiquitous Computing,
GC4: Scalable Ubiquitous Computing Systems, and
GC6: Dependable systems evolution,

and the conclusions for these that can be drawn from our experience. This paper gives a personal view: we cannot here survey the entire field except in the broadest terms, and so, with apologies to its authors, do not discuss the substantial body of related research.

Modelling Complex Systems

Here we take a broad view of distributed computation, encompassing *traditional client-server systems*, *wide-area systems*, *global computing*, *ubiquitous computing*, *peer-to-peer systems*, and *grid computing*.

In all of these, system designers must cope with many highly complex issues.

Some of this complexity is fundamental, arising from the problems of software evolution and version change, partial failure, malicious attack, and mobility (not an exhaustive list!). Other complexity is contingent, arising from historical design choices that are now baked in to the global infrastructure. For example, the *sockets* interface to the TCP and UDP protocols first appeared more that 20 years ago and, while it has changed, it is conceivable that it will never be entirely replaced. On a shorter timescale, the JVM and .NET CLR intermediate languages may be pervasive for a number of years. This combination of complexities makes it extremely difficult to understand systems — well enough to engineer them robustly — with the current software tools and languages, based only on informal intuition and engineering experience. Accordingly, we believe it is necessary to support intuition and experience with rigorous models.

Much theoretical work has been concerned with highly idealised models of systems. This idealisation is essential: for establishing basic concepts and results, to understand design questions in isolation, and to keep the mathematical definitions simple enough that proof is feasible. To be directly applicable, however, we also need models that have very clear relationships to real-world systems, and must consider the integration of design questions in full-system settings. We need semantic descriptions that cover the behaviour of executable code (indeed, of widely deployed and executing code), capturing its response to partial failure, attack, etc.

The construction of these models can be either *post-hoc*, describing an existing software artifact, or *pre-hoc*, during design. The former may be necessary for 'baked in' components of the infrastructure: if, pragmatically, a component cannot be replaced or changed, the best we can do is to describe precisely what its existing behaviour is (exactly which components are in this class depends on the timescale under consideration, of course). The latter can be a powerful design tool, allowing design choices to be described precisely and concisely, and promoting conceptual clarity.

Complete models (of either kind) can be used in several ways: as precise documentation for informal reference, as a basis for rigorous proof of higher layers, and as a basis for applying the many semi-automated analysis and verification techniques — type systems, model-checking, static analysis etc. — either to the executable code or to some well-understood abstraction thereof.

Modelling in Practice

Beginning with the general goal of establishing solid foundations for large-scale distributed computation, we have focussed on a number of specific aspects. This has involved both pre-hoc and post-hoc model building and design, at several levels of abstraction (network protocols, programming languages, and distributed communication and security infrastructure).

Our work on *network semantics* [1, 2, 3] is developing rigorous *post-hoc* behavioural specifications of the ubiquitous UDP and TCP protocols and sockets API, to supplement the existing informal and partial RFCs and texts. The specifications cover interactions across the sockets interface and on the wire; to be useful they must be (and are) closely based on the *de facto* standard — the deployed implementations — rather than the idealisations common in the theoretical literature. They precisely characterise the semantics of partial failure that is visible across the sockets interface, and necessarily deal also with many other details, e.g. of congestion control. The HOL proof assistant is used to sanity-check the large higher-order logic definitions and as a basis for automated symbolic model-checking, to validate the specification by comparing it with traces captured from running implementations (BSD, Linux, and Windows XP). Work on UDP has been completed; a TCP specification is also complete and its validation is in progress.

Our work on *programming languages*, on the other hand, is *pre-hoc*: developing high-level language constructs for distributed programming, expressing them with rigorous semantics and evaluating them with prototype implementations. Early work focussed on mobility. As devices and computations become mobile, there is an increasing need for location-independent communication primitives. To implement these above the low-level location-dependent communication of standard networking requires delicate distributed algorithms. The Nomadic Pict language was developed to express such algorithms (and applications that use them) as clearly as possible. The language was implemented, a range of algorithms with differing properties developed, and one algorithm proven correct – necessitating the development of a rich new semantic theory [4, 5, 6]. Recent work has investigated the broader question of what high-level language support is needed to allow the many high-level abstractions required for distributed programming to be written simply as type-safe libraries in a general-purpose language, above a well-understood TCP/UDP layer. We have primitives for global communication: type-safe (and respecting abstract types), supporting software evolution with versions and version constraints, allowing typed communication between programs with differing versions of modules, and with controlled dynamic rebinding to local resources. No existing languages support all these, or deal with the subtle interactions between modules, type identity, and versions. The Acute prototype language, with a core based on OCaml, has been designed and implemented to test these ideas experimentally [7, 8, 9, 10].

Questions of *security* exist across all levels of abstraction, as attacks are unconstrained. We have studied the *secure encapsulation* of untrusted components [11] and (with current PhD students) are looking at systems for *anonymity and privacy* [12] and *policy- and role-based access control*, the latter with a substantial case study based on the NHS National Electronic Health Record requirements [13].

Addressing the Grand Challenges

Our general goal, establishing solid foundations for large-scale distributed computation, is consistent with those of the three challenges GC2,GC4, and GC6:

GC2 – Science for Global Ubiquitous Computing.

- To develop a coherent informatic science whose concepts, calculi, theories and automated tools allow descriptive and predictive analysis of the GUC at each level of abstraction;
- That every system and software construction – including languages – for the GUC shall employ only these concepts and calculi, and be analysed and justified by these theories and tools.

GC4 – Scalable Ubiquitous Computing Systems.

- A central aim of this challenge for Computer Systems research is to make it feasible to program such systems without having to employ many orders of magnitude more programmers with greater skills than today's software engineers.
- From the model/architectural viewpoint, a goal should be to provide a hierarchy of abstractions that allow us to specify and understand these systems at many levels of detail.

- From the pragmatic viewpoint, rather than a single system implementation, a classical computer science approach suggests itself — a generative approach to building systems, where the standards are the rules that the generator follows, but the systems that it constructs may be quite different for different pieces of the architecture, or different points in the scale.

GC6 – Dependable systems evolution.

- Commercial and industrial-scale software can be developed to be truly dependable, at lower cost and with less development risk than today.
- The vulnerabilities in legacy systems and COTS components can be discovered and corrected, improving their dependability.
- Dependable systems can be evolved dependably including, for a class of applications, just-in-time creation of required services.

Indeed, there is little we would disagree with in any of these proposals. It may be worth commenting on some points of emphasis, however, and on the problems of *scale* and *integration* that arise in this kind of work.

Applied Semantics: Application-Driven Semantics, Semantically-Founded Systems

Primarily, we would like to emphasise the importance of *tightly interlinked* systems and semantic research. System-design problems have become harder, and are now demanding better conceptual and software tools, while theoretical techniques have advanced. It now seems very fruitful to approach the general goal by trying to build particular systems, using whatever theory is required. In doing so, it may be possible to take a model off the shelf, or to use an existing analysis tool, or (the common case, and the ideal for the researcher) one may discover that new semantics, languages or tools must be created, and proceed to do so. Of course, it is vital to choose example systems of the right scale — involving some problems that are not yet well-understood, but not overwhelmingly complex. Further, expectations of timescale must be realistic: if one demands a working system in 3 months (or, indeed, 3 years) then the most useful and creative solutions, which require substantial research, may be automatically ruled out.

The GC2 proposal discusses some possible experimental applications, but they appear to us to be of a longer-term nature — applications for theorists to keep firmly in mind, but perhaps too complex to actually attempt to (re)build them, on new foundations, in the next few years. The complementary GC4 focusses on the new architectural techniques required to build large systems, but does not discuss how they should be expressed — how we can best support good engineering of interfaces and abstraction layers. A short-term goal is to have interesting *fully specified* systems, with integrated rigorous descriptions of the behaviour of communication primitives, programming language, and executable application code above. This is clearly within reach, though dependent on exactly what level of communication is involved (as a step towards this, for example, work is underway by Michael Compton on integrating our UDP/sockets semantics, a semantics for an OCaml fragment, and code written therein, all in the Isabelle proof assistant.) Our personal focus is primarily on improving software quality by providing better, and better-understood, abstractions, not on the complete verification of correctness properties of GC6, but integrated semantic descriptions are a necessary precondition for verification.

We thus see scope for much work in the intersection of GC2, GC4, and GC6. A useful medium-term goal for this is to *demonstrate*, for some particular systems, that semantic tools really can be used to build more them more robustly and more easily, in a way that is persuasive for the software engineer in the street.

The Problems of Scale

In all of this kind of work, one has to deal with problems of *scale* that do not arise when working with small calculi and languages. Firstly, just the amount of detail is large. The description of even a simple real-world protocol or of a moderate-size programming language goes far beyond the point where hand-proof is feasible without prohibitive effort. Indeed, such descriptions are too large even for one to have complete confidence that they are internally type consistent (for example, our TCP/UDP specification is some 14000 lines of HOL; our Acute definition is 35 pages of informal mathematics). Good *tool support* is therefore needed, likely based on existing proof assistants (HOL, Isabelle, Twelf, PVS, Coq, etc.). Automated reasoning has made great strides, but —from our user perspective— much more is required: improved proof automation (e.g. for type preservation proofs), support for object-language syntax (with

variable binding and better surface-syntax support), error reporting, efficiency, and integration between provers to allow portability of definitions.

Secondly, one needs to integrate semantic descriptions of different components (e.g., for us, of TCP/UDP and of OCaml fragments or of Acute). This is not conceptually challenging, but a matter of semantic engineering.

Thirdly, one must have semantics for realistic languages, both for description and for reasoning. Semantic technology and idioms have improved to the point where it is perfectly feasible to design a large language in terms of its operational reduction semantics and type system (though better tool support is needed to manage the detail). Dealing with the number and integration of language features is still challenging, however, and more work is certainly necessary on *compositional semantics* of languages, or on ad-hoc tools for composing definition fragments. Reduction semantics is currently the best definition tool, but for much reasoning a more abstract semantics is necessary: theories of observational congruence, logics, and/or denotational semantics, that simultaneously cover all the language features. For example, we recently wanted to reason about programs in a language with abstract types, concurrency, mutable state, partial failure, and exceptions. Understanding observational congruence for such a combination goes well beyond the state of the art.

Fourthly, one must deal post-hoc with the detail of whatever existing systems are involved in the problem at hand. Better tool support for *testing* conformance of a specification and an implementation is needed, both to validate post-hoc specifications and to provide confidence in implementations of pre-hoc specifications. Such testing cannot provide the level of confidence that complete proof-based verification can, of course, but it is extremely valuable nonetheless, being (largely) automatable, and currently feasible for more complex systems.

Fifthly (specific to programming-language research), there is a major design challenge of selecting and integrating the many language features that have been developed into one or more coherent wholes. For example, many sophisticated type systems for enforcing security properties have been developed, but any attempt at combining all of them would certainly give an unusably-complex language.

Recommendations

We see potential for a real research *community* at the intersection of GC2, GC4, and GC6, marked by significant interaction and collaboration between its members. This synergy could be very productive. However, for it to occur it will be necessary for the present research, publication, and funding culture to change.

1. Most importantly there must be a shift of focus, of both theoretical and practical researchers, towards more tightly-integrated research. This might be encouraged by workshops on large-scale semantics and tool support, which should be satellites of appropriate conferences and be internationally-based, not purely UK-centric. Enlarging the (small) pool of potential researchers with expertise in both systems and semantics could be supported by targetted MSc programmes and PhD summer schools and, to some extent, in undergraduate curricula.

2. The shift of focus must carry over into publication values: the community needs work on large-scale composition, implementation, and tool support just as much as work on core principles and novel theories. The results of this research must be used in anger.

3. It is clear that the scale of problem far exceeds the typical single paper, the work of a single research group over a few years, or even the long-term extended collaboration that has produced e.g. OCaml and Haskell. One should therefore ask what is needed to enable collaborations of this size or larger. We strongly believe they should arise bottom-up — nothing would be worse than recreations of the large industrial standards committees (though in some cases they can be influenced for the better). To encourage the formation of such collaborations sometimes simply longer-term, or larger, funding is required.

4. More subtly, to enable synergy between different research groups, it is desirable for the community to converge somewhat on common notation, common definitions, and common language fragments (contrast, say, the variety of definitions and notation for lambda calculi with the commonality of notation in thermodynamics). All this would enable re-use, e.g. of a semantics for a large Java fragment.

5. Finally, in a virtuous circle, this commonality may be driven by the improved tools which are required. For example, if it were only slightly easier to use automated proof assistants for programming language definition fragments, it might become commonplace to exchange them, and to place them in the public domain.

To conclude, while this is not short-term research, there is urgency. The economic and human cost of poor information infrastructure being built now may be with us for a long time.

Acknowledgements

This paper rests on joint work with many people, including: Mair Allen-Williams, Moritz Becker, Gavin Bierman, Steve Bishop, Gian Luca Cattani, Michael Compton, Matthew Fairbairn, Michael Hicks, James Leifer, Michael Norrish, Gilles Peskine, Benjamin Pierce, Andrei Serjantov, Gareth Stoyle, Asis Unyapoth, Jan Vitek, and Paweł Wojciechowski.

We acknowledge support from a Royal Society University Research Fellowship (Sewell), a St Catharine's College Heller Research Fellowship (Wansbrough), EPSRC grants GRN24872 and GRL62290, and EC FET-GC project IST-2001-33234 PEPITO.

References

[1] Michael Norrish, Peter Sewell, and Keith Wansbrough. Rigour is good for you, and feasible: reflections on formal treatments of C and UDP sockets. In *Proceedings of the 10th ACM SIGOPS European Workshop (Saint-Emilion)*, pages 49–53, September 2002.

[2] Keith Wansbrough, Michael Norrish, Peter Sewell, and Andrei Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proceedings of ESOP 2002: the 11th European Symposium on Programming (Grenoble), LNCS 2305*, pages 278–294, April 2002.

[3] Andrei Serjantov, Peter Sewell, and Keith Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proceedings of TACS 2001: Theoretical Aspects of Computer Software (Sendai), LNCS 2215*, pages 535–559, October 2001.

[4] Asis Unyapoth and Peter Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proceedings of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (London)*, pages 116–127, January 2001.

[5] Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, April–June 2000.

[6] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*, pages 1–31. Springer-Verlag, October 1999.

[7] James Leifer, Peter Sewell, and Keith Wansbrough. Marshalling: Abstraction, rebinding, and version control, 2004. draft, available `http://www.cl.cam.ac.uk/users/pes20`.

[8] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time lambda. In *ICFP: the 8th ACM SIGPLAN International Conference on Functional Programming (Uppsala)*, pages 99–110, August 2003.

[9] James Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proceedings of ICFP 2003: the 8th ACM SIGPLAN International Conference on Functional Programming (Uppsala)*, pages 87–98, August 2003.

[10] Peter Sewell. Modules, abstract types, and distributed versioning. In *POPL: 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (London)*, pages 236–247, January 2001.

[11] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Box-π, wrappers and causality types. *J. Computer Security*, 11(2):135–188, 2003. Invited submission, CSFW 00 special issue.

[12] Andrei Serjantov and Peter Sewell. Passive attack analysis for connection-based anonymity systems. In *ESORICS: European Symp. on Research in Computer Security (Gjøvik)*, pages 116–131, Oct 2003.

[13] Moritz Y. Becker and Peter Sewell. Cassandra: Flexible trust management, applied to electronic health records, 2004. Draft available `http://www.cl.cam.ac.uk/users/mywyb2/`.