Send Statement Considered Harmful, or High Level Coordination Constructs

Phil Trinder

School of Mathematical and Computer Sciences,
Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, U.K.
trinder@macs.hw.ac.uk

Executive Summary

Engineering global or ubiquitous software requires the programmer to specify and reason about the coordination of computations on large-scale dynamic networks. A key part of successful software engineering is to select a language at the appropriate level of abstraction. Currently, while computation languages are well-developed and available at a range of abstraction levels, together with calculi for reasoning about them, coordination is less well developed. Almost all systems are engineered using low level coordination, e.g. using individual sends and receives, analogous to programming with GOTOs. Moreover, only low level calculi are available to support reasoning about coordination. For example the π and ambient calculi only support reasoning at the level of individual operations, e.g. send and receive. Such low level reasoning requires considerable effort, being analogous to reasoning about computation at the level of GOTOs.

The challenge is to develop programming languages with powerful yet high level coordination constructs for global ubiquitous systems (GUS), supported by effective implementations and high level reasoning. Some high level coordination constructs already exist e.g. algorithmic skeletons, but additional constructs are required to address the coordination requirements of global ubiquitous computing. Higher level coordination reasoning will facilitate the construction of dependable GUS, and example high level coordination identities are sketched in section 4 below.

The challenge posed by developing an appropriate level of coordination abstraction is of similar scale to that posed by control abstraction in the 1960s, and data abstraction in the 1980s and 1990s. The new coordination constructs and implementations will form a bridge between two current grand challenges: enabling the calculi developed in the Science for Global Ubiquitous Computing (GC2)to be applied to the practical construction of Ubiquitous Systems (GC4).

Expected advances: Within 5 years we expect to see the design of new high level coordination constructs for GUS. Some of these designs will be implemented, possibly as extensions of existing programming languages. Some GUS demonstrators will be constructed using the new constructs and evaluated in comparison to existing lower level technologies. High level coordination calculi will be developed, possibly as extensions of existing calculi, and used to prove properties of programs written with the new constructs.

1 Introduction

Constructing correct and efficient sequential programming is already a challenging task. Classical parallel, distributed or mobile programming introduces the additional challenge of coordinating computations correctly and effectively. Coordination entails specifying aspects such as process management, communication, synchronisation, resource discovery, and mobility between locations. Global or ubiquitous systems (GUS) introduce additional coordination aspects such as self configuration on dynamic networks and sets of locations.

Five decades of Computer Science have demonstrated that a key part of successful software engineering is to select a language at the appropriate level of abstraction for the task in hand: we wouldn't use an assembly language to build a theorem prover. Computation languages are well-developed and available at a range of levels of abstraction, e.g assemblers, procedural, object-oriented, functional and constraint-logic programming languages. Moreover there are well-developed calculi for specifying and reasoning about computations at different levels of abstraction, e.g. weakest precondition for procedural languages, or equational reasoning for functional languages.

In contrast to the range of computation abstractions available, currently there are only low and intermediate level coordination abstractions, and a lack of appropriate high level coordination abstractions. Moreover existing abstractions are designed to coordinate parallel, distributed and mobile computations, and global ubiquitous computing requires additional coordination, as discussed below.

2 Low and Intermediate Level Coordination

Currently most parallel, distributed and mobile systems are constructed using low level coordination constructs like Sockets, CORBA, OGSA [7] or Globus [6]. There are exceptions, e.g. the MPI library contains some algorithmic skeletons [5]. Low level coordination is very powerful, enabling the programmer to specify arbitrary coordination, and may be essential for some applications.

Constructing software using low level, and to a lesser extent, intermediate level, coordination constructs is highly undesirable for the majority of applications. For example specifying communication using unstructured sends and receives is analogous to specifying computation using unstructured GOTO statements. Moreover using low level constructs places a significant burden on the programmer who must explicitly manage many, often relatively unimportant, coordination details. This in turn encourages programmers to restrict themselves to static, simple or regular coordination. It is hard to reason about programs with low level coordination, not least because of the level of detail. Thus it is hard to be sure that the program is correct, and hence to produce a dependable GUS.

3 High Level Coordination for GUS

Some high level constructs have been developed for parallel and distributed coordination.

- Remote procedure calls (RPCs), or remote method invocation, abstract over a send;compute;receive sequence between client and server.
- Algorithmic skeletons are at a higher level, offering a fixed set of higher order functions, or computational patterns, that combine computation and coordination [3]. For example a parallel map skeleton, *parMap*, typically abstracts over an arbitrary-length sequence of RPCs to different locations.
- Evaluation strategies allow the compositional and higher-order specification of coordination [9]. Not only can almost all algorithmic skeletons be defined, but also application-specific coordination.
- Erlang behaviours abstract over fault tolerant distributed coordination, e.g. the supervisor behaviour is a declarative encapsulation of process monitoring and recovery [1]. We write A *sup (B,C)* p to denote process A monitoring processes B and C using recovery policy p.

Global ubiquitous computing requires additional coordination over those found in a typical parallel, distributed and mobile system. Dynamic networks and sets of locations, and mobile computations and devices require autonomy or self configuration. Trust, security and privacy must be ensured. The scale and duration of GUS will require the automatic management of many of these coordination aspects, and yet it should be possible to manage key aspects of a GUS.

Engineering scalable, dependable and predictable GUS will be facilitated by languages with powerful yet high level coordination constructs, supported by effective trusted and reliable implementations. Such languages will require sophisticated implementation technology to automatically manage low level coordination details. This is a significant challenge: the high-performance community has demonstrated the range and severity of issues with constructing systems that effectively manage even the limited coordination required by parallelism on relatively simple architectures [4].

4 Reasoning about Coordination

Existing calculi like the π and ambient calculi [8, 2] enable reasoning about coordination properties only at a low level, e.g. individual sends and receives. Reasoning at this level is tedious, and analogous to reasoning about computation properties at the level of GOTOs.

The high level coordination constructs should have simple semantics to enable higher level reasoning. High level coordination calculi will include identities familiar from lower level calculi. For example if $\equiv_p$ denotes process network equivalence, and $f\ o||\ g$ denotes parallel function composition where a stream of results from g are piped to f, we have

$$f\ o||\ (g\ o||\ h) \equiv_p (f\ o||\ g)\ o||\ h$$

More importantly the calculi will contain higher level coordination identities, for example the following process network equivalence for the *parMap* algorithmic skeleton from section 3:

$$parMap(f\ o||\ g) \equiv_p (parMap\ f)\ o||\ (parMap\ g)$$

Let us define recovery equivalence, $\equiv_r$, as process network equivalence after recovery from the failure of any process. If processes A, B and C don't interact, and *all* is the recovery policy that kills all supervised processes, then the following identity holds for the Erlang supervisor behaviour from section 3:

$$A\ sup(B, C)\ all \equiv_r A\ sup(C, B)\ all$$

Such high level coordination identities make reasoning about coordination properties far easier. For example it will be far easier to demonstrate that two programs have equivalent process networks. Likewise coordination properties of programs can be relatively easily transformed, e.g. to optimise process placement, and analysed e.g. to identify potentially mobile computations. As with reasoning about computation properties, there is the potential to build tools to automate the reasoning about coordination.

5 Attacking the Challenge

The challenge of developing usable high level coordination abstractions for GUS can be addressed in the following strands.

- *A. Develop new high level coordination constructs* to abstract over GUS coordination aspects, including mobile computation, fault tolerance, trust/ security/privacy, self configuration, information provinence. Existing abstractions will guide the design of constructs for some aspects, but there are currently no high level abstractions for other aspects. Indeed it may not be possible to construct useful high level abstractions for some aspects. The constructs must be both powerful enough to be useful and efficiently implementable.
- *B. Developing high level coordination calculi.* High level coordination reasoning requires coordination calculi over the high level constructs. The high level calculi may be entirely new, or extensions of existing calculi. This strand should interact with strand A, e.g. the high level constructs should be designed to preserve useful identities.
- *C. Construct effective implementations.* The implementations must automatically manage the low level coordination details abstracted in the high level constructs. This will entail both static guarantees of properties, e.g. privacy or security, and also dynamic management of the sets of computations, locations and network connections of a GUS. The implementation must be effective, dependable and secure.
- *D. Tools and methodologies* to support the the engineering of substantial GUS may be developed in the longer term.

References

1. J.L. Armstrong, S.R. Virding, M.C. Williams, and C. Wikstrom. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.
2. L. Cardelli. Mobility and Security. In *Proc. NATO Advanced Study Institute on Foundations of secure Computation*, pages 3–37, Marktoberdorf, Germany, August 1999.
3. M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. PhD thesis, University of Edinburgh, 1988. Also published in book form by Pittman/MIT, 1989.
4. M.I. Cole. Bringing skeletons out of the closet. *Parallel Computing*, 2004. To Appear.
5. Message Passing Interface Forum. Mpi: A message passing intrface standard. *International Journal of Supercomputer Application*, 8(3–4):165–414, 1994.
6. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kufmann, 1999.
7. I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. The physiology of the grid. an open grid services architecture for distributed systems integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 2002. <URL:`http://www.globus.org/research/papers/ogsa.pdf`>.
8. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.
9. P.W. Trinder, K. Hammond, H-W Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.