

Can Philosophy reconcile Computer Science with Software Engineering?

Steve Matthews

Dept. of Computer Science, University of Warwick, Coventry CV4 7AL, UK,
Steve.Matthews@warwick.ac.uk,
www.dcs.warwick.ac.uk/people/staff/Steve.Matthews

Abstract. The discipline of *Computer Science* formalises notions of computable knowledge. However, the ever increasing power of computers leads to ever more useful processing of knowledge that is far from formalisable. If Computer Science is to keep up with computers does it need to mature into *Computer Science Philosophy*?

Relevant aspects of the Autumn Meeting¹

- Philosophical contributions to knowledge representation
- Semantics and knowledge representation
- Evaluation of knowledge: qualitative and quantitative aspects

Semantics and knowledge in Computer Science

The *denotational semantics* of programming languages is an established method in the discipline of Computer Science for proving the consistency of a logic of programs by assigning each program a value in a mathematical structure termed a *domain*. The considerable achievement of the theory of domains and of denotational semantics is to construct a mathematical semantics for self referencing syntactic forms, including apparently absurd equations such as $n = n + 1$. In effect, classical mathematics is extended by introducing values which can be understood as being computable *approximations* to the *truth* of mathematics. In the equation $n = n + 1$, the denotation of n is \perp (pronounced *bottom*), this being the most knowledge that can be computed about any would-be solution to this equation. In an extended algebra of arithmetic, the value of the expression $\perp + 1$ is defined to be \perp . The theory of domains is a highly sophisticated combination of point set topology & lattice theory, and as such leads to a decidedly qualitative semantics for programming languages. However, such modelling of a whole programming language is of little use to the typical programmer who is unlikely to be a gifted pure mathematician. More likely, a practice-driven engineering-minded individual who appreciates programs in terms of their *behaviour*, which often need only be approximately understood.

¹ This document is a statement of interest (SOI) for the Autumn Meeting 2005 of the *SIG Philosophy and Informatics*, held as *KI 2005 WS-3 : Philosophy and Artificial Intelligence*, Koblenz, Germany, 11/9/05.

Lessons from Software Engineering

If, as we propose it can, denotational semantics is to become of more relevance to Software Engineering, its notion of *approximation* has to be reconciled with well established counterparts in Engineering. For example, $\pi = 3.142$ is a provably false mathematical proposition, yet nonetheless a highly useful approximating statement in Engineering practice. In denotational semantics $3.141_$ is the output so far from an ongoing computation, a *partial* number so to speak that is consistent with π , and could *become* $3.1415_$, then $3.14159_$, and so on. Yet such *approximations* as *partial values* are rarely found in Software Engineering? Why not? Denotational semantics is sufficient for proving logical consistency, but lacks Engineering notions of semantics. Software engineers, being engineers, routinely simplify their work by approximating to within, perhaps poorly articulated, margins of error. Their work can be usefully *reliable*, but possibly not well definable in mathematics as being *correct*. When their work is in theory definable as being mathematically *correct* it may, in practice, be too large or complex for anyone to be able to completely know its correctness.

Computer Science Philosophy?

And so, the useful notion of *reliability* in Software Engineering is not always well defined in the form of mathematics which is Computer Science, while the ideal notion of software *correctness* in Computer Science is often unobtainable in practice in Software Engineering. Yet surely these notions of *correctness* and *reliability* have important understandable overlap? Is it possible that *reliability* could be expressed within a more general but rigorous notion of *correctness*? For such a generalisation to be no more than a precise description of imprecision would not go far enough. For example, how fuzzy can fuzzy logic actually be if it, as it does, assigns a real number to be the precise definition of the truth extent of each imprecise truth value? Such fundamental discussion of *correctness* in Computer Science versus *reliability* in Software Engineering is, we argue, necessary and ontological, but beyond the strict confines of Computer Science. The term *Computer Science Philosophy* is coined in this SOI for any rigorous endeavour to extend the borders of Computer Science to embrace necessary non mathematical notions found in Software Engineering such as reliability.

The development of Computer Science over the past fifty years has been impressive by any standard. Denotational semantics demonstrates how Computer Science attempts to model imprecise knowledge within its own discipline. However, such imprecision as precise mathematics is often of little use to software engineers in practice. Imprecision is discussed in this SOI merely to illustrate a philosophical challenge of reconciling Computer Science with Software Engineering. In addition, *time*, *non determinism*, and, *experimentation & improvement in software development* also pose interesting ontological challenges for Computer Science. Can the discipline of Philosophy help Computer Science to rigorously understand & extend its own borders in order to model semantics that Software Engineering knows to be essential & reliable?