## DESCRIPTION: A NEGLECTED TOPIC IN SE (ORIGINAL TITLE)

## DEFINING A DISCIPLINE OF DESCRIPTION (PUBLISHED TITLE)

At a 1985 Symposium on Unintentional Nuclear War, Brian Cantwell Smith argued that the notion of correctness for computer software is inherently limited: "Just because a program is 'proven correct' ... you cannot be sure that it will do what you intend."

A fundamental part of his argument was that in a computer-based system the computer — the hardware plus the software — interacts with the real world. To understand the system we must understand both the computer and the world, and how they interact. This understanding depends on some kind of model of the world. Some developers express the model in the mathematical notations of a formal method; others in the notations of structured or object-oriented methods. This model participates in two relationships: with the computer on one side, and with the world on the other. We are well equipped to analyse and understand the first relationship — the correspondence between the computer and the model. But we have no theory of the second relationship — the correspondence between the model and the real world.

What Cantwell Smith identified as the lack of a theory is a symptom of something far more urgent and far more practical than the word *theory* suggests. We have neglected a basic discipline: the construction and use of well-engineered descriptions of an *informal* reality. *Well-engineered* means descriptions that are precise and reliable enough for specific practical purposes. In Cantwell Smith's terms, these descriptions of reality are the model; a well-engineered description is a model that corresponds in the desired way to the real world.

### *The Problem of Description*

Description in this sense is a practical discipline in its own right. We all understand the need for an important element of this discipline in a commonplace situation. Someone stops in a passing car and asks for directions. You don't say "continue until you reach a slight bend, then turn into the road with a rather poor surface and go on to the corner where there is an attractive house". How will the unfortunate traveller know what is a "slight bend", or "rather poor surface", or an "attractive house"? If you mean to be helpful you give directions in terms of features that will be unambiguously recognised: "turn left at the second traffic lights, then right at Joe's Restaurant".

You are trying to solve the problem of giving an unambiguous description of an informal reality. It's difficult, because an informal reality has unbounded resources for defeating your best efforts. Perhaps at the second traffic lights there are two exits that could count as "turning left". Perhaps the sign visible from the road actually reads "Joe's Bar". Perhaps the first traffic lights operate only during the rush hours. Will they be seen if they are not lit, or — if they are seen — will the driver count them as traffic lights? The right descriptive technique can help you to avoid these pitfalls. There's a serious technical problem here, if you want to take it seriously.

### *The Formal Computer and the Informal World*

In software engineering the problem appears in a heightened form. The computer with its software is formal — the physical nature of the hardware does not taint the precision and predictability of its programmed behaviour. The interpretation of a program is always unambiguous.

But the system of which the computer is a part, and the purposes it is intended to serve in the world, are informal. This is why requirements engineering is so hard. It demands descriptions of an informal world precise enough to be intelligibly related to the formal behaviour of the computer and its software, and reliable enough to guarantee that the formal behaviour will evoke the required results in the informal world. Failure in requirements engineering is often identified as the cause of system failure: the computer and its software behaved exactly as specified, but the result was a disaster.

The designers of an avionics system needed to ensure that reverse thrust was disabled except when the plane has touched down on landing. They reasoned convincingly that whenever the plane is landing and has already touched down, its weight is on the landing wheels and the wheels are rolling on the runway. But one plane using this system landed on a poorly drained runway in a rainstorm. The wheels were aquaplaning, not rolling; the system disabled reverse thrust; and the plane ran off the end of the runway. The aquaplaning possibility had not been considered. In an informal world there are always factors that have not been considered that might, just possibly, invalidate the analysis and its conclusions.

*Is the World Our Business?*

It is tempting to react to this difficulty by excluding requirements engineering and its irksomely informal subject matter from the scope of software engineering. Let the application domain experts write the requirements. Let them specify the external behaviour demanded of the computer. We, as software engineers, will be concerned only with designing and building the software to ensure that behaviour. Our responsibility begins at the heart of the computer and ends at its input-output interface.

But this is not a sensible choice. A software specification that is rigorously restricted to behaviour at the external interface is usually unintelligible. In a system to control traffic lights at a road junction, the interface consists only of the signals input to the machine (from pedestrian push buttons) and the control signals output by the machine (to the devices that turn the lights on and off). But the required relationship among these signals is intelligible only in the context of the road junction layout, the speeds of vehicles using the junction, and the permitted and possible behaviour of pedestrians. None of this is directly visible at the interface. If we want our problems to make any kind of sense, we must deal squarely with the informal worlds in which they are located.

Further, we have a moral responsibility. Computers endow the systems of which they are a part with distinctive and potentially unwelcome characteristics. The computer is behaviourally more complex than any other component in the system. Its formality makes this complexity unyielding and inexorable, and therefore extremely dangerous. Unlike a manual system, a computer-based system allows no appeal to common sense until the damage has been done. At the same time, highly automated business and administrative procedures and embedded control systems are becoming more and more commonplace and more and more ambitious. The application domain experts, accustomed to systems with a substantial element of overriding human control, need the direct and wholehearted involvement of software engineers to help them to master the ever-growing complexity of modern systems and their effects.

A discipline of description must be at the heart of this mastery. Yet it has been sadly neglected. Being concerned just with the meeting place of the formal and the informal,

it has fallen through the gap between the more formal disciplines –- programming, formal specification methods and discrete mathematics –- and the various informal disciplines –- sociological, ethnographic, economic, managerial and many others –- that have found a place in software engineering.

*A Discipline of Description*

What are the elements of this discipline? Here is my personal list of some topics that I think important:–

- Designations. The basis of any description of reality is a choice of the ground terms to be used and a careful explanation of what phenomena each one denotes in the reality described. You are choosing a ground term when you prefer 'traffic lights' to 'slight bend'; you are explaining its denotation when you say exactly how to recognise 'traffic lights'. A designation gives the syntax of the formal term and the corresponding explanation. A full set of designations for a description is what logicians call an 'interpretation' (but, being logicians, they leave out most of the explanations).
- Definition. It is vital to understand and exploit the difference between designation and definition. If you have *designated* the event classes 'issue *m* widgets' and 'receive *n* widgets', you may want to turn your attention to the question 'how many widgets are in stock?' There are essentially two things you can do. You can *define* 'widget stock' as the cumulative total of widgets received minus the cumulative total of widgets issued in your designated events. Or you could *designate* 'widget stock' as the number of widgets actually in the bin in the store when you go to look. The two are fundamentally different.
- Bypassing faulty domain terminology. You can't take all your designations from the standard terminology of the application domain. That terminology has usually evolved in an essentially informal environment, where its inadequacies are easily handled by commonsense exceptional procedures. Airline check-in agents are not confused by the notion of a 'flight', although one flight may involve two different airplanes and two distinct flights might be combined in one journey of one airplane. But it's impossible to make a good designation of a 'flight' in this informal sense. You have to bypass this kind of informal term by definitions built on top of soundly designated terms.
- Error analysis. In an informal world, a designation can be no more than an approximation, with an error that must be estimated. For the avionics system the designations 'landing' and 'wheels rolling' were such approximations. When you use these approximate terms to reason about the world, the error can build up just as it does when you use floating-point arithmetic to calculate over real quantities. The designated terms, and the use to be made of them, must be chosen so that the resulting error will be small enough to give a sufficiently reliable system.
- Description structuring. It's easy to introduce confusion by poor large-scale structuring of descriptions. One example is confusion between those properties and behaviour that the computer must impart to the world, and those properties and behaviour that the world possesses regardless of the computer. A lift can not move from the first to the third floor without passing the second floor, regardless of the computer's behaviour. But the property of stopping at floors whose request buttons have been pressed must be imparted by the computer. The grammatical distinction between 'shall' and 'will' is an attempt to capture this distinction of meaning. But

the distinction can not be adequately captured in this localised way; more powerful structuring techniques are necessary.

- Recognising the limitations of formalisms. Dijkstra famously observed that in the pursuit of program correctness "testing can show the presence of errors but not their absence". In the pursuit of well-engineered descriptions of the real world we should recognise — and every student of software engineering should be taught — that formalisation can show the presence of description errors, but not their absence.

*A Neglected Discipline*

The neglect of a discipline of description can be seen in many software engineering courses; evidence of the accompanying neglect in software engineering practice appears regularly in Peter Neumann's Risks Forum. We are not alone in practising this kind of neglect. In a wonderful book about mechanical and structural engineering Eugene S Ferguson writes:

"The real problem of engineering education is the implicit acceptance of the notion that high-status analytical courses are superior to those that encourage the student to develop an intuitive 'feel' for the incalculable complexity of engineering practice in the real world."

A discipline of description would be concerned precisely with this "incalculable complexity" of the real world. If we pay it the attention it deserves our customers, and the users of our systems, will have cause to thank us. And passing visitors needing directions should be grateful too.

*References:*

Brian Cantwell Smith; The Limits of Correctness; a paper prepared for the Symposium on Unintentional Nuclear War, Fifth Congress of the International Physicians for the Prevention of Nuclear War, Budapest, Hungary, June 28 – July 1 1985.

Eugene S Ferguson; Engineering and the Mind's Eye. MIT 1992, p168.

[1980 words including references and headings]

[IEEE Software 15,5 pp14-17, September/October 1998]