The United Nations University

UNU-IIST

International Institute for Software Technology

UTP and sustainability

Yifeng Chen and J. W. Sanders

August 2010

UNU-IIST Report No. 440

R

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research $\boxed{\text{R}}$, Technical $\boxed{\text{T}}$, Compendia $\boxed{\text{C}}$ or Administrative $\boxed{\text{A}}$. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

Peter Haddawy, Director

The United Nations
University

UNU-IIST

International Institute for
Software Technology

P.O. Box 3058
Macao

UTP and sustainability

Yifeng Chen and J. W. Sanders

Abstract

Hoare and He's approach to unifying theories of programming, UTP, is a dozen years old. In spite of the importance of its ideas, UTP does not seem to be attracting due interest. The purpose of this article is to discuss why that is the case, and to consider UTP's destiny. To do so it analyses the nature of UTP, focusing primarily on unification, and makes suggestions to expand its use.

Yifeng Chen is a Research Professor at the HCST Key Lab at the School of EECS, Peking University, China. Previously he spent several years in the U.K., as Senior Lecturer at the University of Durham and Lecturer at the University of Leicester, after completing a *DPhil.* at the University of Oxford. His interests include imperative, parallel and object-oriented programming languages, including design, translation, static analysis, semantics, specifications and their support for decentralised software development.

Jeff Sanders is Principal Research Fellow at UNU-IIST. His interests lie largely in Formal Methods.

The first author was partially supported by the China HGJ Significant Project 2009ZX01036-001-002-4, and the second author by the Macao Science and Technology Development Fund, under the PEARL project grant number 041/2007/A3.

Copyright © **2010** by UNU-IIST

Contents

1 Preamble

The history of science is a maze of roads not taken; of ideas not pursued. You can't drift in style from Shanghai to Beijing in a Zeppelin. Nor do you explain the evolution of the Panda using Lamarckism. The Theremin has not replaced the Stadivarius. Your laptop bears little resemblance to Babbage's difference engine. Those analogue computers, the astrolabe, slide rule and Bush's differential analyser, have all been interred in a graveyard for nondigital devices. And wither quantum computing?[1]

It is interesting to speculate on the reasons for lack of success. They may be social: too much has already been invested in alternatives (the world has overlooked *nonstandard calculus*, in spite of its capturing the Leibnizian intuition of infinitesimals and boasting a first-year textbook [20]). Or the reasons may be commercial: a more powerful competitor has its own alternative or an alternative offers better profitability (VHS quickly dominated Betamax [40]). Of course most often the reasons are simply scientific (perpetual motion machines, the geocentric solar system (or was it universe?) and phlogiston).

Science evolves by following pathways at the expense of those neglected, for whatever reasons. Seldom does a choice between paths have the opportunity to be weighed up publicly.

Is UTP a dead-end? The purpose of this paper is to reflect on and promote discussion of just that question. The importunate reader might skip Section 2, in which the problem confronting UTP is considered; Section 3, in which UTP is 'kick started'; Section 4, in which alternative UTP projects are considered; and instead move straight to the Conclusions.

2 UTP at the crossroads

2.1 The evidence

UTP is struggling. It seems that the previous two conferences (UTP2006 in Durham [9] and UTP2008 in Dublin) required considerable organisational skill on the parts of Steve Dunne and Andrew Butterfield respectively. Recall the difficulty attracting interest in the present event, despite Shengchao Chin's best efforts. And witness the continued poor acceptance rate for papers. How many Ph.D. theses has UTP supported? Are there case studies that make non-specialists want to use it? Are there special conference tracks that have the effect of incorporating UTP into the wider community?

For comparison, think of the manner in which Z [12, 38] became established: the early case study of IBM's CICS [18]; the large number of M.Sc. and Ph.D. theses (from Oxford alone); its integration into the wider community of Formal Methods and the blossoming of case studies; tool support and its adoption by industry; organisation of user-group meetings (which owe much to Jonathan Bowen); expansion and re-use (for instance object-Z [8]); and the proliferation of courses and books. A similar

[1]The reader is invited to a complementary parlour game: list ever-more insignificant things which nonetheless prevail. Let's start with '*lorem ipsum*'.

story could be told for the theories VDM, CSP, CCS, ...

It is now a dozen years since the UTP textbook [17] was published. Since Z started with a whimper rather than a bang, comparison is difficult; perhaps for Z a similar time would have elapsed by the mid 90s. By then it appeared stronger and to be expanding *much* more rapidly than UTP does now. Surely the time has come to reflect on the situation.

Let's start, well, at the beginning. What might be expected of theories of programming? Why seek to unify them? Finally, how might unification be expected to go?

2.2 Theories of programming

Once upon a time, in the early days of ALGOL, a theory of programming consisted of the syntax of a programming language, an advance that has been accorded the name BNF for John Bachus and Peter Nauer. That 'theory' helped programmers who, at a time when new programming languages were appearing fast, furious and in a wide range of styles, could otherwise learn a new language only by following examples.

But compiler writers required more: a semantics by which to validate and compare their products. At first, semantic descriptions were informal. The case of ALGOL 60 provoked the transition to formality: its reports [30, 3] of 1960 and 1963 in natural language were (inevitably?) criticised as being ambiguous [21]. The variety of semantic approaches was evident from the start: an axiomatic semantics was given to Pascal [14] in 1971; an operational style was used for PL/I [23] in 1971 and ALGOL 68 [39] in 1975; and denotational semantics was demonstrated on ALGOL-like languages [25] by 1976. A theory of programming consisted, by the mid 1970's, of a language's syntax accompanied by a semantic description. It seems fair to say that the unfortunate divergence between programming and theoretical computer science dates from this time.

During the first half of the 1970's programmers, now in the rôle of humble software engineers, required yet more from a theory of programming. Support was required for *system engineering*: for verification against a more asbtract specification of a design that was either posited, or obtained by incremental development in a manner supporting the top-down approach of engineering

$$\mathit{Spec} = \mathit{Design}_0 \sqsubseteq \mathit{Design}_1 \sqsubseteq \ldots \sqsubseteq \mathit{Design}_n = \mathit{Impl}$$

(including algorithmic refinements over the same state space, and refinements over distinct spaces by data representation). The important feature of the 'domain of discourse' is that it be powerful enough to express specifications, code, and the combinations that arise at intermediate levels of design. The semantic model is thus required to span various levels of abstraction and to be founded on a (reflexive and) transitive notion of refinement.

That approach has since retained its importance because it enables:

- a design to be verified against a specification (without an understanding of conformance, what does a specification mean?);
- abstract interpretation and model checking, firstly of the abstract model and secondly of the property being checked against it;
- comprehension of a system incrementally, by layers of successively finer detail; that approach has been traditionally used qualitatively to describe complex software (for example operating systems [22]), and now is able to be interpreted quantitatively;
- stepwise system derivation, of the kind begun by Dijkstra in the 1970's for simple programs but now extended to systems, through all layers of abstraction using laws and machine assistance;
- comprehension of new behaviours (like concurrency, probability and time) when intuition alone is too risky as a basis for programming;
- program analysis of the usual kinds: data-flow, constraint-based, abstract interpretation, type systems and effect systems [32].

This time theoreticians took longer to respond, though the first step was immediate. In 1975 Dijkstra provided the predicate transformer model [7], then at just one level of abstration and without explicit use of laws or the refinement relation $\sqsubseteq$. Over the next fifteen years the concepts Dijkstra had introduced for code were extended to the more general commands appropriate for software engineering (unbounded nondeterminism, angelic choice and unenabled commands ('miracles') [31, 28, 1, 29], and data refinement [35]) and studied in both the predicate-transformer and binary-relation models [15]. The result was, by 1990, what two decades later is still recognised as a 'theory of programming':

1. a semantic domain $(\mathcal{X}, \sqsubseteq)$, incorporating a partial order representing refinement
2. a mapping $[\![\cdot]\!]$ from the syntactically-defined 'programming' language to the semantic domain $\mathcal{X}$
3. accompanying laws that are sound (and ideally complete) with respect to the semantic model.

2.3 Unification

But whilst theoretical support for such theories of programming is satisfactory, applications remain scant. Examples include: functional programming languages [27] (cartesian-closed categories); the guarded-command language [15] (predicate transformers or binary relations); process algebras CSP [37] (failures and divergences) and CCS [26] (transition trees); and receptive-process theory (for use in asynchronous devices) [19] (failures and divergences).

The difficulty is that, as new features are incorporated, the complexity increases due to the interaction between the new feature and (potentially) *each* existing feature. The incorporation of probability with nondeterminism [24] is certainly a success; but occam [36] has been of limited success semantically, due to interactions between state, nondeterminism and synchronisation.

Fortunately in many paradigms of computation, a new feature interacts in a severely circumscribed manner with previous features. So there is hope that a satisfactory theory can be obtained incrementally, by adding new features gradually to existing theory.

That, of course, is what is meant by *unification* in UTP, and why the approach is so vitally important. Without it there seems little chance of providing a patently correct, comprehensible, semantics for something like occam which combines, as already observed, various features that interact nontrivially. Without it that list of successes seems destined to remain short. But using it, one might hope to describe occam, for example, in layers that correspond to sequential programs, nondeterministic programs, reactive programs, and finally occam processes. Indeed that has been one of the principle motivations for UTP, and a measure by which its success can be judged.

To demonstrate its utility, a unification of theories of programming ought to explicate further pressing paradigms of computation. Examples include

- service computing
- real time
- object orientation (including mutable objects)
- component-based systems
- adaptivity and other self-$*$ system properties
- hybrid and cyberphysical systems
- machine learning
- quantum computation
- game-theoretic semantics
- hardware systems
- biologically-inspired systems.

If the unifying approach *is* as important as has just been reasoned from a scientific viewpoint, why has it not been more widely adopted in a dozen years?

2.4 The three-chapter problem

It has been observed that many students of UTP do not progress past Chapter 3 of Hoare and He's eleven-chapter text [17]. The implication is that by being exposed to only the first thirty percent of the book (85 pages of 282), their view of UTP is dominated by relations, predicates and the healthiness conditions

Hi, for $1 \leq \mathbf{i} \leq 4$. Indeed there does seem to be evidence, amongst students and even researchers, for the accuracy of this harsh claim.

Can the 'three-chapter problem' be related to our lack of progress in unifying theories? Can that in turn be part of the reason behind the limited adoption of UTP? Since there seems little hope of systematic progress in the area of providing theories of contemporary programming without use of unification, some investigation seems required.

2.5 What might might be expected of unification?

It has been argued in Section 2.2 that a theory of 'programming' consists of a semantic domain (partially ordered), an interpretation of the language in that domain and a collection of sound laws for the language constructs. The purpose of this section is, in view of the UTP programme having seemingly stalled, to consider afresh what might be expected from unification.

The expectation is that, by viewing theories hierarchically, a simple theory $\mathcal{A}$ is to be embedded in a more complex $\mathcal{C}$ in a manner that enables $\mathcal{A}$'s semantics to be imported. So the semantics of the more complex $\mathcal{C}$, as far as it concerns just the features it shares with the simple language $\mathcal{A}$, has already been provided in $\mathcal{A}$; only features unique to $\mathcal{C}$ (lying outside the range of the embedding) need now be considered. The theory of $\mathcal{C}$ has been unified with that of $\mathcal{A}$ *via* the embedding. Examples will be familiar to UTP *aficionados*; several are considered in Section 3.

The partial order $\sqsubseteq$ of each theory captures conformance. As usual there are operators corresponding to its *infimum*, $\sqcap$, and *supremum*, $\sqcup$. The former arises from abstraction, or information hiding, *via* local blocks and it is preserved by the embedding ε—as is required for lifting a semantics that includes $\sqcap$—iff for any family $\mathcal{E}$ (empty, nonempty and finite, or infinite) in the abstract domain $\mathcal{A}$,

(1) $\varepsilon.\sqcap\mathcal{E} \;=\; \sqcap\{\varepsilon.E \mid E \in \mathcal{E}\}\,.$

That condition is equivalent to ε being the embedding in an embedding-projection pair (ε,π) known as a *Galois connection* and defined by the equivalence

(2) $\varepsilon.a \sqsubseteq_{\mathcal{C}} c \;\equiv\; a \sqsubseteq_{\mathcal{A}} \pi.c\,.$

The case in which ε is injective embeds $\mathcal{A}$ in $\mathcal{C}$ and so forms the basis of the hierarchical approach. Then the connection is called a *Galois embedding*; see Figure 1.

So Galois connections and embeddings must be expected to play a central rôle in unifying theories. Moreover, in lifting semantics to a more detailed level, the embedding ε must preserve further combinators. For instance preservation of sequential composition,

$$\varepsilon.(r \mathbin{;} s) \;=\; \varepsilon.r \mathbin{;} \varepsilon.s\,,$$

$$\begin{tikzcd}
a \sqsubseteq \pi.c \ \text{in} \ (\mathcal{A}, \sqsubseteq_{\mathcal{A}}) \arrow[d, bend right, "\varepsilon"'] \\
\varepsilon.a \sqsubseteq c \ \text{in} \ (\mathcal{C}, \sqsubseteq_{\mathcal{C}}) \arrow[u, bend right, "\pi"']
\end{tikzcd}$$

Figure 1: A Galois embedding $gc(\varepsilon, \pi; \mathcal{A}, \mathcal{C})$ relates abstract and concrete theories.

enables ε to be used to lift the behaviour of a sequential composition from one level to the next in the hierarchy. Similarly for the other combinators, including recursion (and hence) iteration. That way, laws in $\mathcal{A}$ are *re-used* in $\mathcal{C}$: a major benefit of unifying theories.

It will then be important to characterise the lifted space *ran*.ε as a subset of $\mathcal{C}$ and furthermore to determine—if possible—the manner in which it generates $\mathcal{C}$. For that determines the $\mathcal{C}$-semantics in terms of the $\mathcal{A}$-semantics.

But now we find ourselves firmly in Chapter 4, without having mentioned predicates or healthiness conditions. How can that be? From our present viewpoint, predicates merely form the basis of certain models (predicate transformers, for example!); and healthiness conditions are merely used in defining domains. Suddenly both have disappeared from the centre of the stage on which we expect unification to be performed.

Is it possible that there is an alternative entry to UTP which starts from Galois connections—Chapter 4—instead of Chapter 3? If so, how does it go and what then is the importance of Chapter 3? Perhaps by following it the 'three-chapter problem' might be avoided.

3 A fresh start

3.1 Computability: theory $\mathcal{P}$

The theory of Computer Science began in the 1930s with the various models, by Hilbert, Turing, Kleene, Church, Markov *et al.*, of the concept of a *computation* (or of recursiveness). By modelling a 'mechanism', mathematicians had for the first time to model the possibility of nontermination. Today such computations are called *predeterministic* because from any initial state they are either nonterminating or deterministic, and written using syntax like that of Figure 2. The set of all predeterministic programs over state space X is written $Predet(X)$.

abort	nontermination
$x := e$	assignment, with expression e
P **if** b **else** Q	conditional
$P \fatsemi Q$	sequential composition
$\mu.F$	recursion

Figure 2: Syntax for the space *Predet* of predeterministic programs. Assignment is assumed to be predeterministic and recursion to be with respect to a continuous function.

Our intention is that sequential composition be associative with identity the assignment **skip** (which changes no variable). More interestingly, if a nonterminating program precedes or follows another program the result remains nonterminating:

(3) $\quad \mathbf{abort} \fatsemi P \;=\; \mathbf{abort} \;=\; P \fatsemi \mathbf{abort}\,.$

Of course recursion includes iteration as tail recursion.

The time-honoured model [5] for $Predet(X)$ consists of partial functions on X, with refinement as extension; it is denoted $\mathcal{P}(X)$:

$$\mathcal{P}(X) \;:=\; (X \nrightarrow X, \supseteq)\,.$$

It is a domain[2] with least element the empty partial function, $\{\}$, with maximal elements the total functions and with compact elements the partial functions having finite domains. The semantic mapping is given in Figure 3. Both its well-definedness, and soundness of the laws, are routine.

By starting with the 'historical' theory $\mathcal{P}(X)$, explicit consideration of healthiness conditions has been avoided: predeterminism is captured enitrely by the type $X \nrightarrow X$.

3.2 An alternative: theory $\mathcal{Q}$

A popular alternative, in view of models soon to come, is to replace the partial functions with total functions whose range includes a 'virtual' element for nontermination. Thus each partial function is made total on X by mapping each element outside its domain to the virtual element $\bot$. Furthermore, for the new model to be closed under sequential composition, it must be 'homogeneous': the virtual state $\bot$

[2] By 'domain' here is meant a complete partial order in which each element is the supremum of its compact approximations. Recall that an element k is *compact* iff any directed set $\mathcal{E}$ that exceeds it contains an element which does so: if $k \sqsubseteq \sqcup\mathcal{E}$ then $\exists e : \mathcal{E} \cdot k \sqsubseteq e$. In the case of partial functions, the domain conditions mean: $\forall f : \mathcal{P} \cdot f = \cap\{k : \mathcal{P} \mid \#(dom.k) < \infty \wedge k \sqsubseteq f\}$. Indeed without loss of generality there k ranges over singleton partial functions: $\#(dom.k) = 1$.

$$\begin{array}{rcl}
[\![\textbf{abort}]\!]_{\mathcal{P}} & := & \{\} \\
[\![x := e]\!]_{\mathcal{P}} & := & \lambda x : X \cdot e \\
[\![P \textbf{ if } b \textbf{ else } Q]\!]_{\mathcal{P}} & := & \lambda x : X \cdot [\![P]\!]_{\mathcal{P}}.x \textbf{ if } b.x \textbf{ else } [\![Q]\!]_{\mathcal{P}}.x \\
[\![P \mathbin{;} Q]\!]_{\mathcal{P}} & := & [\![Q]\!]_{\mathcal{P}} \circ [\![P]\!]_{\mathcal{P}} \\
[\![\mu.F]\!]_{\mathcal{P}} & := & \cup\{f : \mathcal{P}(X) \mid F.f \subseteq f\}
\end{array}$$

Figure 3: The $\mathcal{P}$ semantics of predeterministic programs, in which program P is denoted by a partial function $[\![P]\!]_{\mathcal{P}}$ and variable x is used for both its argument and the state of the program. Recursion is the least fixed point of F, as given by the first recursion theorem of Kleene (for instance [5], Theorem 10.3.1).

must also belong to the domain. Let

$$X_\perp := X \cup \{\perp\}.$$

Now for the left zero law in (3) to hold, it suffices for each denotation f of a program to be *strict* (with the flat ordering on $X_\perp$):

$$f. \perp = \perp .$$

For the right zero law in (3) to hold, it suffices for f also to be *up-closed* at that bottom element. Such relational behaviour is most easily captured by defining, for a function $f : X \to X$, its '(relational) strict and up-closed extension to $X_\perp$' by

$$(f)_\perp := f \cup \{\perp\} \times X_\perp$$

(an idea that is extended from functions f to relations in Section 3.3).

Writing $pre.f$ for the set of elements of X not mapped by the extension f to $\perp$,

$$pre.f := \{x:X \mid f.x \neq \perp\},$$

in order for ε to be isotone, the partial order of conformance must translate in the new model to:

$$f \sqsubseteq f' := (f \restriction pre.f = f' \restriction pre.f).$$

Thus

$$\mathcal{Q}(X) := (\{(f)_\perp \mid \exists f : X \to X\}, \sqsubseteq),$$

$$\begin{tikzcd}
& \cdot \arrow[dl, "{[\![\,\cdot\,]\!]_{\mathcal{P}}}"'] \arrow[dr, "{[\![\,\cdot\,]\!]_{\mathcal{Q}}}"] & \\
\mathcal{P}\ \cdot \arrow[rr, dotted, "\varepsilon"'] & & \cdot\ \mathcal{Q}
\end{tikzcd}$$

Figure 4: Using ε to translate the semantics of predeterministic programs from $\mathcal{P}$ to $\mathcal{Q}$.

and the translation function is

$$\begin{aligned}
&\varepsilon : \mathcal{P}(X) \to \mathcal{Q}(X)\\
&\varepsilon.f \ := \ f \cup \{\, (x, \perp) \mid x \in X_\perp \setminus dom.f \,\}\,.
\end{aligned}$$

Theorem 1 The translation function $\varepsilon : \mathcal{P}(X) \to \mathcal{Q}(X)$

1. is an isotone bijection so that in particular $ran.\varepsilon$ is the carrier of $\mathcal{Q}(X)$;
2. ensures that the domain $\mathcal{Q}(X)$ has least element the constant function $\perp$, maximal elements the functions f with $pre.f = X$ and compact elements the functions f with $pre.f$ finite;
3. preserves total functions (*i.e.* assignment): if f is total then $(\varepsilon.f) \restriction X \ = \ f$;
4. preserves composition: $\varepsilon.(f \circ g) \ = \ (\varepsilon.f) \circ (\varepsilon.g)$.

Now the $\mathcal{Q}$ semantics of *Predet* is obtained by translating the $\mathcal{P}$ semantics with the embedding ε as indicated in Figure 4. Theorem 1 ensures accuracy of the result and preservation of the laws; the result is given in Figure 5.

It is worth emphasising that the semantics is not defined anew, but translated by ε from $\mathcal{P}$. For example

$$\begin{aligned}
&[\![\textbf{abort} \,;\, P]\!]_{\mathcal{Q}}\\
= \quad &\text{definition of } \mathcal{Q} \text{ semantics, Figure 4}\\
&\varepsilon.[\![\textbf{abort} \,;\, P]\!]_{\mathcal{P}}\\
= \quad &\text{law of } \mathcal{P} \text{ semantics}\\
&\varepsilon.[\![\textbf{abort}]\!]_{\mathcal{P}}\\
= \quad &\text{definition of } \mathcal{Q} \text{ semantics again}\\
&[\![\textbf{abort}]\!]_{\mathcal{Q}}\,.
\end{aligned}$$

3.3 Nondeterminism: theory $\mathcal{D}$

Nondeterminism arises for several reasons. Firstly, it might simply be inherent in functionality being specified: locate x in an array (where x may occur more than once); find a minimum spanning tree

$$\begin{array}{rcl}
[\![\textbf{abort}]\!]_{\mathcal{Q}} & = & \lambda x : X \cdot \bot \\
[\![x := e]\!]_{\mathcal{Q}} & = & \varepsilon.(\lambda x : X \cdot e) \\
[\![P \textbf{ if } b \textbf{ else } Q]\!]_{\mathcal{Q}} & = & \lambda x : X \cdot [\![P]\!]_{\mathcal{Q}}.x \textbf{ if } b.x \textbf{ else } [\![Q]\!]_{\mathcal{Q}}.x \\
[\![P \mathbin{;} Q]\!]_{\mathcal{Q}} & = & [\![Q]\!]_{\mathcal{Q}} \circ [\![P]\!]_{\mathcal{Q}} \\
[\![\mu.F]\!]_{\mathcal{Q}} & = & \sqcup\{f : \mathcal{Q}(X) \mid F.f \sqsubseteq f\}
\end{array}$$

Figure 5: The $\mathcal{Q}$ semantics of predeterministic programs, inferred from the $\mathcal{P}$ semantics (Figure 3) using the technique of Figure 4.

(where there may be several), a shortest path, a Hamiltonian circuit, ... Secondly, it might be the result of abstracting the mechanism determining a choice made at a lower level of abstraction: a random-number generator whose seed and mechanism of generation are concealed. Thirdly, it might be assumed in order to ensure that reasoning is local: a choice determined by testing a global variable might be assumed to be a nondeterministic choice in order to avoid global reasoning.

Predeterministic programs are extended to be finitely[3] nondeterministic by augmenting the language *Predet* with a binary combinator for *nondeterministic choice*:

$$P \sqcap Q \,.$$

The set of such programs over state space X is written $Prog(X)$ and the relationship of conformance still written $\sqsubseteq$. Its connection with nondeterminism is, as already observed,

$$P \sqsubseteq P' \;\equiv\; P \sqcap P' = P \,.$$

Important laws involving programs and nondeterminism include:

$$\begin{array}{llcl}
(4) & P \sqcap \textbf{abort} & = & \textbf{abort} \\
(5) & (P \sqcap Q) \mathbin{;} R & = & (P \mathbin{;} R) \sqcap (Q \mathbin{;} R) \\
(6) & P \mathbin{;} (Q \sqcap R) & = & (P \mathbin{;} Q) \sqcap (P \mathbin{;} R) \,.
\end{array}$$

The first characterises the Dijkstra-Hoare approach: in order to guarantee entirely correct implementations, a theory must ensure that the (nondeterministic) possibility of an error is identified with certain error. In (5) the demonic choice responsible for the nondeterminism is made first on both sides, which are therefore indistinguishable. Law (6) is more subtle because the choice is made first on the right, but on the left only after P; nonetheless, the two programs are expected to have identical behaviour (because

[3]More precisely, the nondeterministic choice is now considered of any nonempty finite set of programs. That is equivalent to the nondeterministic choice of two programs, by induction and the laws of associativity, idempotence and commutativity of binary nondeterministic choice.

P on the left-hand side is a program and not a more general kind of computation (like angelic choice) able to offers behaviour which the later demonic choice can exploit).

What is the relationship between $Prog(X)$ and $Predet(X)$, *i.e.* between programs and predeterministic programs? The following law 'quantifies' the relationship by expressing each program as the (not necessarily finitely) nondeterministic combination of its predeterministic refinements.

(7) $\forall P : Prog(X) \cdot P \;=\; \sqcap\{\, Q : Predet(X) \mid P \sqsubseteq Q \,\}$

A 'dual' law, extended from predeterministic programs to programs and hence analogous to the 'domain law' in Footnote 2, expresses each program (and so in particular, each predeterministic program) as the supremum of the compact programs (defined in that footnote, and to be characterised semantically in Theorem 2) it refines:

(8) $\forall Q : Prog(X) \cdot Q \;=\; \sqcup\{\, K : Prog(X) \mid\; K \sqsubseteq Q\,,\; K \text{ compact} \,\}\,.$

A compelling model of nondeterministic programs [15] consists of allowing the elements of the model $\mathcal{Q}(X)$ to be multivalued, since $\mathcal{Q}(X)$ already captures nontermination with the value $\perp$, and now would capture nondeterminism as multi-valueness of a relation. Then the partial order of conformance, 'at least as deterministic as', between such relations would be containment (as sets)

$$r \sqsubseteq s \;\;\equiv\;\; r \supseteq s\,.$$

Let us determine the healthiness conditions on such a relation r on $X_\perp$, using the same method as for $\mathcal{Q}(X)$. For the right zero law to hold in (3), it suffices for r to be total on X, as were the elements of $\mathcal{Q}(X)$

(9) $\forall x : X \cdot \exists x' : X_\perp \cdot x r x'\,.$

For the left zero law, again it suffices for r to map $\perp$ to all of $X_\perp$.

In order for **abort** to be minimum in the refinement ordering of containment, Law (4), it suffices for

$$x r \perp \;\Rightarrow\; \forall x' : X_\perp \cdot x r x'\,.$$

Finally in order for the least upper bound, or intersection, of a chain of healthy relations again to be healthy, as required for recursion, it suffices for the image of each state to be *finitary*: to be either all of $X_\perp$ or nonempty and finite

(10) $\{x' : X_\perp \mid x r x'\} \;\neq\; X_\perp \;\;\Rightarrow\;\; 0 < \#\{x' : X_\perp \mid x r x'\} < \infty\,.$

Evidently being nonempty supersedes totality (9).

Those conditions can be abbreviated using the notation $X \leftrightarrow X$ for the type of all relations on X and $r.(\!|\, x \,|\!)$ for the relational image of r at x

$$r.(\!|\, x \,|\!) \ := \ \{x' : X_\perp \mid x r x'\}\,.$$

That model is called $\mathcal{D}(X)$, and has ordering $\supseteq$ and carrier set

$$\{r : X_\perp \leftrightarrow X_\perp \mid \left(\begin{array}{l} \perp r \perp \\ \forall x : X_\perp \cdot \left(\begin{array}{l} x r \perp \Rightarrow r.(\!|\, x \,|\!) = X_\perp \\ r.(\!|\, x \,|\!) \neq X_\perp \ \Rightarrow \ 0 < \#r.(\!|\, x \,|\!) < \infty \end{array}\right)\end{array}\right)\}\,.$$

There is a Galois connection from the model $\mathcal{Q}(X)$ for predeterministic programs to the model $\mathcal{D}(X)$, whose embedding is

$$\begin{array}{ll} & \varepsilon : \mathcal{Q}(X) \to \mathcal{D}(X) \\ (11) \qquad & \varepsilon.f \ := \ f \cup (X_\perp \backslash pre.f) \times X_\perp\,. \end{array}$$

In other words,

$$x\,(\varepsilon.f)\,x' \ \equiv \ (x \in pre.f \ \Rightarrow \ f.x = x')\,.$$

Its adjoint $\pi.r$ denotes the largest partial function in r which 'accounts for all of r's results at its arguments'. It may be thought of as the largest partial function which approximates, in $\mathcal{Q}(X)$, total relation r. Indeed that is the form for π expected by adjunction:

$$\pi.r \ = \ \cup\{f : \mathcal{Q}(X) \mid \varepsilon.f \supseteq r\}\,.$$

Then:

Theorem 2 The function $\varepsilon : \mathcal{Q}(X) \to \mathcal{D}(X)$

1. is an injection that preserves arbitrary suprema from $\mathcal{Q}(X)$ under $\sqsubseteq$ to $\mathcal{D}(X)$ under $\supseteq$: more generally, Definition (11) of ε makes sense if its argument is merely a relation, and then for *any* subset F of the carrier of $\mathcal{Q}(X)$ (not just those having a well-defined supremum $\sqcup F \in \mathcal{Q}(X)$),

$$\varepsilon.\cup F \ = \ \cap\{\varepsilon.f \mid f \in F\}\,;$$

2. has adjoint $\pi : \mathcal{D}(X) \to \mathcal{Q}(X)$, thus $gc(\varepsilon, \pi;\ \mathcal{Q}(X), \mathcal{D}(X))$, where

   (12) $\pi.r \ = \ \{(x,y) : r \mid y \neq \bot \ \wedge \ \forall x' \neq \bot \cdot x r x' \Rightarrow x' = y\}$

   which is $(\supseteq, \sqsubseteq)$-continuous: if R is a $\supseteq$-directed subset of $\mathcal{D}(X)$ then

   (13) $\pi.\cap R \ = \ \sqcup\{\pi.r \mid r \in R\}$;

3. has range which generates $\mathcal{D}(X)$ under nonempty finite unions:

   $$\mathcal{D}(X) \ = \ \{\cup F \mid F \subseteq \mathit{ran}.\varepsilon \ \text{ is nonempty and finite}\};$$

4. ensures that the domain $\mathcal{D}(X)$ has least element the universal relation on $X_\bot$, maximal elements the (total) functions and compact elements the relations r with $\mathit{pre}.r$ finite (extending Definition (4) from functions to relations); thus each $r : \mathcal{D}$ is the supremum of compact elements which it refines (a fact which is weaker than 3 since each compact element of $\mathcal{D}$ is a nonempty finite union of elements of ran ε);

5. preserves sequential composition: $\varepsilon.(\mathrm{id}_X) = (\mathrm{id}_X)_\bot$ and $\varepsilon.(f \circ g) \ = \ (\varepsilon.g) \mathbin{⨾} (\varepsilon.f)$.

It is convenient to define an embedding from relations on X to those on $X_\bot$ to capture that part of the healthiness conditions relating to initial virtual state:

$$\begin{aligned}&(\,\cdot\,)_\bot : (X \leftrightarrow X) \to (X_\bot \leftrightarrow X_\bot)\\ &(r)_\bot \ = \ r \cup \{\bot\} \times X_\bot\,.\end{aligned}$$

Since $(\,\cdot\,)_\bot$ preserves arbitrary intersections (though only nonempty unions) it is Galois from $(X \leftrightarrow X, \supseteq)$ to $(X_\bot \leftrightarrow X_\bot, \supseteq)$. Its adjoint is restriction to X :

$$\begin{aligned}&\pi : (X_\bot \leftrightarrow X_\bot) \to (X \leftrightarrow X)\\ &\pi.s \ := \ s \cap (X \times X)\,,\end{aligned}$$

a projection that preserves arbitrary intersections (as well as arbitrary unions as expected from the basic property of Galois connections) and is surjective. The embedding $(\,\cdot\,)_\bot$ is injective (as expected from properties of Galois connections) and preserves sequential composition:

(14) $(r \mathbin{⨾} s)_\bot \ = \ (r)_\bot \mathbin{⨾} (s)_\bot$.

The semantic space $\mathcal{D}(X)$ is comprehensively more complex than $\mathcal{P}(X)$. Our task, then, is to define the semantics of $\mathit{Prog}(X)$ in $\mathcal{D}(X)$ in such a way that the simplicity of the $\mathcal{P}(X)$ semantics is not obscured. That is achieved—of course—by lifting with ε *via* $\mathcal{Q}(X)$.

$$\begin{array}{rcl}
[\![\textbf{abort}]\!]_{\mathcal{D}} & = & X_\perp \times X_\perp \\
[\![x := e]\!]_{\mathcal{D}} & = & (\lambda x : X \cdot e)_\perp \\
[\![P \textbf{ if } b \textbf{ else } Q]\!]_{\mathcal{D}} & = & \{\, (x,x') \mid x[\![P]\!]_{\mathcal{D}}x' \textbf{ if } b.x \textbf{ else } x[\![Q]\!]_{\mathcal{D}}x' \,\} \\
[\![P \mathbin{;} Q]\!]_{\mathcal{D}} & = & [\![P]\!]_{\mathcal{D}} \mathbin{;} [\![Q]\!]_{\mathcal{D}} \\
[\![\mu.F]\!]_{\mathcal{D}} & = & \cap\{\, d : \mathcal{D}(X) \mid F.d \supseteq d \,\} \\
[\![P \sqcap Q]\!]_{\mathcal{D}} & = & [\![P]\!]_{\mathcal{D}} \cup [\![Q]\!]_{\mathcal{D}}
\end{array}$$

Figure 6: Important properties of the relational semantics for *Prog*. Function F is monotone on $\mathcal{D}$.

For each $P : Prog(X)$ its relational semantics $[\![P]\!]_{\mathcal{D}}$ is defined by Law (7) using union for nondeterministic choice and the lifting (Figure 4), under the Galois connection of Theorem 2, of the $\mathcal{Q}$ semantics of P's predeterministic refinements:

$$(15) \quad [\![P]\!]_{\mathcal{D}} \;=\; \cup\{\, \varepsilon.[\![Q]\!]_{\mathcal{P}} \mid Q \in Predet(X) \;\wedge\; P \sqsubseteq Q \,\}\,.$$

In particular, if P is itself predeterministic then

$$[\![P]\!]_{\mathcal{D}} \;=\; \varepsilon.[\![P]\!]_{\mathcal{Q}}\,.$$

For example **skip**, because it is deterministic, has semantics

$[\![\textbf{skip}]\!]_{\mathcal{D}}$

= definition of $\mathcal{D}$ semantics

$\varepsilon.[\![\textbf{skip}]\!]_{\mathcal{Q}}$

= $\mathcal{Q}$ semantics with **skip** abbreviating $(x := x)$

$\varepsilon.(\lambda x : X \cdot x)$

= definition of ε

$(\lambda x : X \cdot x)_\perp\,.$

A similar argument works for **abort**; as does the fact that $[\![\textbf{abort}]\!]_{\mathcal{Q}}$ is the least element of $\mathcal{Q}(X)$ and ε preserves minima (a basic property of Galois connections).

Thus the $\mathcal{D}$ semantics of $Prog(X)$ is defined by lifting on $Predet(X)$ and otherwise by union. Now the properties, that before were a matter of definition in the $\mathcal{P}$ semantics of Figure 3, are simply inferred, though with a little more work than for the $\mathcal{Q}$ semantics as inferred in Figure 5; see Figure 6.

Consider, for example, sequential composition. The proof relies on predeterministic computations whose $\mathcal{P}$ semantics consists of a singleton partial function (recall Footnote 2); thus the computation terminates from just a single state. Writing $Predet_1(X)$ for the set of such computations, for $P, P' : Prog(X)$,

$$
\begin{array}{ll}
[\![P \,\mathrm{;}\, P']\!]_{\mathcal{D}} & \\
= & (15) \\
\cup\{\varepsilon.[\![R]\!]_{\mathcal{Q}} \mid R \in \mathit{Predet}(X) \wedge P \,\mathrm{;}\, P' \sqsubseteq R\} & \\
= & \text{Footnote 2 and set theory} \\
\cup\{\varepsilon.[R]_{\mathcal{Q}} \mid R \in \mathit{Predet}_1(X) \wedge P \,\mathrm{;}\, P' \sqsubseteq R\} & \\
= & \text{property of } \mathit{Predet}_1(X) \\
\cup\{\varepsilon.[\![R]\!]_{\mathcal{Q}} \mid \exists Q, Q' \in \mathit{Predet}_1(X) \wedge P \sqsubseteq Q \wedge P' \sqsubseteq Q' \wedge R = Q \,\mathrm{;}\, Q'\} & \\
= & \text{1-point law} \\
\cup\{\varepsilon.[\![Q \,\mathrm{;}\, Q']\!]_{\mathcal{Q}} \mid Q, Q' \in \mathit{Predet}_1(X) \wedge P \sqsubseteq Q \wedge P' \sqsubseteq Q'\} & \\
= & \varepsilon \text{ preserves sequential composition (Theorem 2, Part 5)} \\
\cup\{\varepsilon.[\![Q]\!]_{\mathcal{Q}} \,\mathrm{;}\, \varepsilon.[\![Q']\!]_{\mathcal{Q}} \mid Q, Q' \in \mathit{Predet}_1(X) \wedge P \sqsubseteq Q \wedge P' \sqsubseteq Q'\} & \\
= & \text{set theory} \\
\quad \cup\{\varepsilon.[\![Q]\!]_{\mathcal{Q}} \mid Q \in \mathit{Predet}_1(X) \wedge P \sqsubseteq Q\} \,\mathrm{;} & \\
\quad \cup\{\varepsilon.[\![Q']\!]_{\mathcal{Q}} \mid Q' \in \mathit{Predet}_1(X) \wedge P' \sqsubseteq Q'\} & \\
= & \text{Footnote 2 again} \\
\quad \cup\{\varepsilon.[\![Q]\!]_{\mathcal{Q}} \mid Q \in \mathit{Predet}(X) \wedge P \sqsubseteq Q\} \,\mathrm{;} & \\
\quad \cup\{\varepsilon.[\![Q']\!]_{\mathcal{Q}} \mid Q' \in \mathit{Predet}(X) \wedge P' \sqsubseteq Q'\} & \\
= & (15) \\
[\![P]\!]_{\mathcal{D}} \,\mathrm{;}\, [\![P']\!]_{\mathcal{D}} \,. &
\end{array}
$$

The case of nondeterminism is similar using instead (in the third step) the property that, for $Q : \mathit{Predet}_1(X)$,

$$P \sqcap P' \sqsubseteq Q \;\equiv\; P \sqsubseteq Q \vee P' \sqsubseteq Q\,.$$

The proofs of Laws (4) to (6) are immediate from basic set theory.

There is an alternative to this approach to the semantics of $\mathit{Prog}(X)$ based on Law (7) with $\cup$ for nondeterminism. It assigns semantics by structural induction on $P : \mathit{Prog}(X)$, 'building in' Equation (15) at each step. But then Law (7) must be checked and so the amount of work is equivalent. The former approach has been chosen because it seems to extend better to more complex domains, like probabilistic domains.

In summary, a Galois connection has been used to lift the $\mathcal{Q}$ semantics, and laws, to $\mathcal{D}$.

3.4 Angelic choice: theory $\mathcal{T}$

Just as Software Engineering brought to light (demonic) nondeterminism, so the formal development process discussed in Section 2.2 revealed the utility of 'partially enabled' computations and 'angelic' choice. We call such computations, which extend programs, *commands*.

magic	the command that is never enabled
$\sqcap \mathcal{F}$	nondeterminstic choice over $\mathcal{F}$
$\sqcup \mathcal{F}$	angelic choice over $\mathcal{F}$

Figure 7: Syntax completing the space $Comm(X)$ of commands over state space X: the unenabled command, and arbitrary nondeterministic and angelic choices. $\mathcal{F}$ is an arbitrary set of commands.

An example of a partially-enabled command is choice of an element from a set which happens to be empty; computation cannot be started—is not enabled—in a manner that is dual to a computation that fails to terminate. This situation arises when a procedure for choosing an element from a set is used in a context which ensures the set is nonempty; but when developed 'in isolation', the empty case must also be considered.

Angelic choice is simply supremum $\sqcup$, the dual of nondeterminism $\sqcap$. A simple example is provided by the angelic choice of two consistent commands. The first, R, chooses x nondeterministically between 0 and 1 whilst the second, S, chooses nondeterministically between 1 and 2. Their angelic choice $R \sqcup S$ is the weakest program stronger than both: $x := 1$.

If R and S had not been consistent in that example then their angelic choice, their supremum, would not have been a program. The supremum of an inconsistent set of commands is a command (though not a program) that is never enabled. Notation for the command that is never enabled and for angelic choice are introduced in Figure 7, as is our last ingredient of command space: arbitrary (rather than just binary) nondeterminism. The set of commands on X is written $Comm(X)$. As usual, the relation of conformance is $\sqsubseteq$, satisfying (4). Of course equivalently:

$$P \sqsubseteq P' \quad \equiv \quad P \sqcup P' = P' .$$

With the extension from programs to commands, the previous laws must be revisited for correctness. Law (5) remains valid: the nondeterministic choice is made initially on both sides and so the demon resolving the nondeterminism, confronted with the same choices, produces the same behaviours. But for just that reason its partner (6) does not remain valid, and must be weakened: for $R, S, T : Comm(X)$,

(16) $R \mathbin{;} (S \sqcap T) \ \sqsubseteq \ (R \mathbin{;} S) \sqcap (R \mathbin{;} T)$.

Refinement there must of course hold by monotonicity. But equality may fail since the demon (having memory but not prescience), has more choices the later it acts. There are thus fewer choices on the right and so fewer behaviours than on the left. The choices coincide if execution of R results in no angelic choice by which the demon might profit: if R is free of angelic choice.

Important laws involving the new combinators include:

(17) $R \sqcup \mathbf{magic} \quad = \quad \mathbf{magic}$

(18) $\mathbf{magic} \mathbin{;} R \;=\; \mathbf{magic}$
(19) $(R \sqcup S) \mathbin{;} T \;=\; (R \mathbin{;} T) \sqcup (S \mathbin{;} T)$
(20) $R \mathbin{;} (S \sqcup T) \;\sqsupseteq\; (R \mathbin{;} S) \sqcup (R \mathbin{;} T)\,.$

The first, (17), says that **magic** is indeed dual to **abort** and so is the greatest (or 'most refined') command (and thus equals the empty angelic choice $\sqcap\{\,\}$). The second says that an unenabled command cannot be enabled by any sequential successor (even **abort**). In (19) the choice is made initially on both sides so, reasoning as above (with the angel in place of the demon), equality holds. But (20) is dual to (16): on the right the angel acts early and—having prescience but not memory—has more choices and so produces more behaviours; alternatively, the refinement follows by monotonicity. The choices coincide if execution of *R* results in no nondeterministic choices by which the angel might profit: if *R* is predeterministic.

The relationship between commands and programs is given by the law analogous to (8) (evidently the analogue of (7) fails): for any command *R*

(21) $R \;=\; \sqcup\{P : \mathit{Prog}(X) \mid P \sqsubseteq R\}\,.$

In fact the domain property holds: without loss of generality, program *P* can be assumed to be compact.

In the relational model $\mathcal{D}(X)$, angelic choice must be intersection and partial enabledness must therefore be captured by partial-ness of a relation. But that means the healthiness condition of totality, (9), no longer holds. Because nondeterminism is now arbitrary, the finitary condition (10) also fails (at both ends of the inequality, in view of lack of totality). Thus all that remains is strictness and upclosure. The extension to $\mathcal{D}(X)$ consisting of relations satisfying just strictness and upclosure, but with the same criterion of conformance, is called $\mathcal{R}(X)$.

The space $\mathcal{R}(X)$ is a domain and a complete lattice with same least element as $\mathcal{D}(X)$ but greatest element $(\{\,\})_\perp$ and compact elements the cofinite 'subsets' of $X_\perp \times X_\perp$. Moreover it is a Boolean algebra under the complement $r \mapsto (X_\perp \times X_\perp \setminus r)_\perp$. However the natural embedding of $\mathcal{D}(X)$ in $\mathcal{R}(X)$ is not Galois. Otherwise its adjoint π would map the greatest element in $\mathcal{R}(X)$ to a greatest element of $\mathcal{D}(X)$; but no such element exists.[4]

Nonetheless the injection of $\mathcal{D}(X)$ in $\mathcal{R}(X)$ does generate $\mathcal{R}(X)$ under arbitrary intersections, reflecting Law (21) (recall that from Theorem 2 nonempty finite unions were used to generate $\mathcal{D}(X)$ from $\mathcal{Q}(X)$, reflecting Law (7)). Thus the carrier set of $\mathcal{R}(X)$ equals

$$\{\cap F \mid F \subseteq \mathcal{D}(X)\}\,.$$

The relational semantics of *Comm*(X) may be thought of—like the semantics for *Prog*—as follows.

[4]Since the natural embedding from $\mathcal{D}(X)$ to $\mathcal{R}(X)$ preserves arbitrary unions, why is it not Galois by adjunction? Because suprema in $\mathcal{R}(X)$ (arbitrary unions) are not the same as suprema in $\mathcal{D}(X)$; consider for example the empty union.

$$
\begin{array}{rcl}
[\![\mathbf{magic}]\!]_{\mathcal{R}} & := & (\{\})_{\perp} \\
[\![\sqcap \mathcal{F}]\!]_{\mathcal{R}} & := & \cup\{\, [\![P]\!]_{\mathcal{R}} \mid P \in \mathcal{F} \,\} \\
[\![\sqcup \mathcal{F}]\!]_{\mathcal{R}} & := & \cap\{\, [\![P]\!]_{\mathcal{R}} \mid P \in \mathcal{F} \,\}
\end{array}
$$

Figure 8: Relational semantics for $Comm(X)$; this augments the extension of the semantics in Equation (15) from $\mathcal{D}$ to $\mathcal{R}$ using the natural embedding.

1. Firstly, the $\mathcal{R}$ semantics equals the $\mathcal{D}$ semantics for commands that are code (like **skip**). In other words the $\mathcal{R}$ semantics *extends* the $\mathcal{D}$ semantics.
2. Secondly, the $\mathcal{R}$ semantics is inferred from the $\mathcal{D}$ semantics by extending the combinators of code to commands (as in the case of sequential composition, or even arbitrary nondeterministic choice, on *Prog* from *Predet*). This is possible because the natural embedding preserves those combinators.
3. Thirdly, it is defined for the (new) combinator of angelic choice by edict, to be intersection.

Thus the $\mathcal{R}$ semantics of *Comm* is provided by Equation (15) (thus extended) and Figure 8 (which also includes arbitrary nondeterminism and its empty case, **magic**).

The proofs of Laws (17), (18) and (20) are now straightforward using basic set theory. For example, for Law (20),

$$
\begin{array}{ll}
[\![P \mathbin{;} (Q \sqcup R)]\!]_{\mathcal{R}} & \\
= & \mathcal{R} \text{ semantics of } \sqcup \text{ and } ; \text{ from Figure 8} \\
[\![P]\!]_{\mathcal{R}} \mathbin{;} ([\![Q]\!]_{\mathcal{R}} \cap [\![R]\!]_{\mathcal{R}}) & \\
\subseteq & \text{set theory} \\
([\![P]\!]_{\mathcal{R}} \mathbin{;} [\![Q]\!]_{\mathcal{R}}) \cap ([\![P]\!]_{\mathcal{R}} \mathbin{;} [\![R]\!]_{\mathcal{R}}) & \\
= & \mathcal{R} \text{ semantics of } ; \text{ and } \sqcup \text{ again} \\
[\![(P \mathbin{;} Q) \sqcup (P \mathbin{;} R)]\!]_{\mathcal{R}} \,. &
\end{array}
$$

Moreover equality holds in the middle step if, pointwise, the relation $[\![P]\!]_{\mathcal{R}}$ either maps to $\perp$ (and hence to all of $X_{\perp}$) or is single valued: as required, the command P is predeterministic.

Unfortunately, for Identity (19) the analogous argument establishes only $\sqsupseteq$, unless relation $[\![R]\!]_{\mathcal{R}}$ is a total function; in other words, command R is deterministic. Furthermore in Law (16) equality always holds (the existential quantification of $;$ distributing the $\cup$ of nondeterminism). It is inferred that the relational model $\mathcal{R}(X)$ does not fully capture angelic behaviour.

Thus stretching the relational model $\mathcal{R}(X)$ from programs to commands reveals deficiencies. The situation is analogous to the introduction of nondeterminism: the model $\mathcal{P}(X)$ was simply not expressive enough and so was extended to $\mathcal{D}(X)$. Now with the introduction of angelic choice, the relational model is in turn not expressive enough and must be extended.

Again, a more detailed model is needed. One possibility is the 'binary multirelation' model [33] of Rewitzky. Instead the *premier* model of sequential semantics, Dijkstra's predicate-transformer model, is chosen.

The *predicate-transformer* model (Dijkstra [7]) views each command as transforming postconditions (predicates on final states) to preconditions (predicates on initial states). For command P, the operational interpretation of its transformer semantics $[\![P]\!]_{\mathcal{T}}$ is: for any postcondition q and any initial state x

$$[\![P]\!]_{\mathcal{T}}.q.x \text{ holds iff } P \text{ terminates from } x \text{ in a state satisfying } q\,.$$

Of course that is sufficient to motivate a formal definition of the semantics. But our interest here lies in reusing the relational semantics to infer the transformer semantics, as far as that is possible.

Let $(pred.X, \leq)$ denote the space of all predicates (*i.e.* conditions) on X partially ordered by implication. The *predicate-transformer* model, $\mathcal{T}(X)$, of commands consists of those predicate transformers $t : pred.X \to pred.X$ that are *monotone*

$$q \leq q' \;\Rightarrow\; t.q \leq t.q'\,,$$

ordered under the lifting of the ordering on predicates

$$t \leq t' \;:=\; \forall q : pred.X \cdot t.q \leq t'.q\,.$$

Then $\mathcal{T}(X)$ is a domain and complete lattice with least and greatest elements the constant functions *false* and *true* respectively. Its compact elements are the transformers t for which there is a finite subset $F \subseteq X$ such that

(22) $\forall q : pred.X \cdot t.q \;=\; \vee\{q.x \mid x \in F\}\,.$

The space $\mathcal{T}(X)$ is endowed with an involution (see Back and von Wright [2])

$$t^*.q \;:= \neg t.\neg q$$

that preserves sequential composition but exchanges nondeterministic with angelic choice, enabledness with termination and **magic** with **abort**.

The embedding from relations $\mathcal{R}(X)$ to transformers $\mathcal{T}(X)$ is traditionally called the *weakest precondition*

$$\begin{array}{l} wp : \mathcal{R}(X) \to \mathcal{T}(X) \\ wp.r.q.x \;:=\; \forall x' : X_\perp \cdot x\,r\,x' \;\Rightarrow\; (x' \neq \perp \wedge q.x')\,. \end{array}$$

It is Galois, but with orders reversed. Writing $(\mathcal{A}, \leq)^\sim$ for $(\mathcal{A}, \geq)$,

Theorem 3 The function $wp : \mathcal{R}(X) \to \mathcal{T}(X)$

1. is an injection that preserves arbitrary suprema from $\mathcal{R}^\sim$ to $\mathcal{T}^\sim$: for any subset $R \subseteq \mathcal{R}$,

   (23) $wp.\cup R \;=\; \wedge\{wp.r \mid r \in R\}$;

2. has adjoint the *relational projection*, $rp : \mathcal{T}(X) \to \mathcal{R}(X)$ completing a Galois connection: $gc(wp, rp; \mathcal{R}^\sim, \mathcal{T}^\sim)$, where

   (24) $x\,(rp.t)\,x' \;:=\; x = \perp \;\vee\; \forall q : pred.X \cdot t.q.x \Rightarrow q.x'$

   which of course preserves infima: for any subset T of the carrier of $\mathcal{T}(X)$,

   (25) $rp.\vee T \;=\; \cap\{rp.t \mid t \in T\}$

   but moreover preserves suprema:

   (26) $rp.\wedge T \;=\; \cup\{rp.t \mid t \in T\}$;

3. satisfies merely

   (27) $wp.(r \cap s) \;\geq\; (wp.r) \vee (wp.s)$

   rather than equality (in contrast to the identities (23), (25) and (26));

4. has range $ran.wp$ consisting of the conjunctive transformers,

   (28) $t \in ran.wp \;\equiv\; \forall q, q' : pred.X \cdot t.(q \wedge q') = t.q \wedge t.q'$,

   and that generates the carrier of $\mathcal{T}(X)$ under angelic choice:[5]

   (29) $\mathcal{T}(X) \;=\; \{\vee F \mid F \subseteq ran.wp\}$;

[5]The space $\mathcal{T}(X)$ is also generated by the composition of *wp* with its involution [2]—$\forall t \cdot \exists u, v : ran.wp \cdot t = u^* \circ v$—but that fact appears less useful here because the transformer involution is not the lifting of an involution on relations [34].

5. ensures that the domain $\mathcal{T}(X)^\sim$ has least element the constant function $\lambda q : pred.X \cdot true$, greatest element the constant function $\lambda q : pred.X \cdot false$ and with compact elements the transformers analogous (because of the reversal of orders) to those described in (22);
6. preserves sequential composition: $wp.(\mathrm{id}_X)_\perp = \mathrm{id}_{pred.X}$ and $wp.(r \mathbin{;} s) \; = \; (wp.r) \circ (wp.s)$, as does its adjoint *rp* in the reverse direction.

As expected from adjunction, the projection *rp.t* defined by (24) is the largest relation that approximates *t* under *wp* .

The semantic space $\mathcal{T}(X)$ appears deceptively simple although the manner of expressing a computation is radically different from that in relations. Naturally a Galois connection is used to bridge the gap!

The Galois connection can be used to lift much of the relational semantics to transformers following our standard approach. As usual for Galois connections, it maps the least element $(\{\})_\perp$ in $\mathcal{R}(X)^\sim$ to the least element, the constant transformer *true*, in $\mathcal{T}(X)^\sim$ thus providing the semantics of **magic**. For sequential composition,

$$\begin{array}{ll}
[\![P \mathbin{;} Q]\!]_\mathcal{T} & \\
= & \text{definition of } \mathcal{T} \text{ semantics} \\
wp.[\![P \mathbin{;} Q]\!]_\mathcal{R} & \\
= & \text{definition of } \mathcal{R} \text{ semantics, Equation (15) and Figure 8} \\
wp.([\![P]\!]_\mathcal{R} \mathbin{;} [\![Q]\!]_\mathcal{R}) & \\
= & \text{property of } wp\text{, Theorem 3.6} \\
(wp.[\![P]\!]_\mathcal{R}) \circ (wp.[\![Q]\!]_\mathcal{R}) & \\
= & \text{definition of } \mathcal{T} \text{ semantics again} \\
[\![P]\!]_\mathcal{T} \circ [\![Q]\!]_\mathcal{T} \,. &
\end{array}$$

It maps arbitrary unions in $\mathcal{R}(X)$ to arbitrary conjunctions in $\mathcal{T}(X)$, by (23), thus providing the semantics of arbitrary nondeterminism. But the lack of equality in (27) means that *wp* can not be used to lift angelic choice from $\mathcal{R}(X)$ to $\mathcal{T}(X)$. That must simply be defined to be disjunction. The resulting transformer semantics is given in Figure 9.

The proofs of Laws (17), (18) and (19) are now straightforward using elementary logic. For Law (20),

$$\begin{array}{ll}
[\![P \mathbin{;} (Q \sqcup R)]\!]_\mathcal{T} & \\
= & \mathcal{T} \text{ semantics of } \sqcup \text{ and } ; \text{ from Figure 9} \\
[\![P]\!]_\mathcal{T} \circ ([\![Q]\!]_\mathcal{T} \vee [\![R]\!]_\mathcal{T}) & \\
\geq & \text{monotonicity} \\
([\![P]\!]_\mathcal{T} \circ [\![Q]\!]_\mathcal{T}) \vee ([\![P]\!]_\mathcal{T} \circ [\![R]\!]_\mathcal{T}) & \\
= & \mathcal{T} \text{ semantics of } ; \text{ and } \sqcup \text{ again}
\end{array}$$

$$\begin{array}{rcl}
[\![\mathbf{abort}]\!]_{\mathcal{T}} & := & \mathit{false} \\
[\![\mathbf{magic}]\!]_{\mathcal{T}} & := & \mathit{true} \\
[\![x := e]\!]_{\mathcal{T}} & := & \lambda q : pred.X \cdot q[e/x] \\
[\![P \textbf{ if } b \textbf{ else } Q]\!]_{\mathcal{T}} & := & [\![P]\!]_{\mathcal{T}} \textbf{ if } b \textbf{ else } [\![Q]\!]_{\mathcal{T}} \\
[\![P \mathbin{;} Q]\!]_{\mathcal{T}} & := & [\![P]\!]_{\mathcal{T}} \circ [\![Q]\!]_{\mathcal{T}} \\
[\![\mu.F]\!]_{\mathcal{T}} & := & \vee\{ t : \mathcal{T}(X) \mid F.t \leq t \} \\
[\![\sqcap \mathcal{F}]\!]_{\mathcal{T}} & := & \wedge\{ [\![P]\!]_{\mathcal{T}} \mid P \in \mathcal{F} \} \\
[\![\sqcup \mathcal{F}]\!]_{\mathcal{T}} & := & \vee\{ [\![P]\!]_{\mathcal{T}} \mid P \in \mathcal{F} \}
\end{array}$$

Figure 9: Transformer semantics for commands, inferred from Figure 8 using the *wp* Galois connection.

$$[\![(P \mathbin{;} Q) \sqcup (P \mathbin{;} R)]\!]_{\mathcal{T}} .$$

Moreover equality holds in the middle step if the transformer $[\![P]\!]_{\mathcal{T}}$ is disjunctive; in other words, the command P is predeterministic.

3.5 Refinement calculus

For theoretical purposes a computation is conveniently described as a single predicate; a form familiar to this audience is $(p \wedge ok) \Rightarrow (P \wedge ok')$. Similarly for the purposes of specification; a familiar form is the body of a Z specification [38]. But for development towards code, it is more convenient to reveal the *precondition*, or predicate from which termination is assured. That idea, first promoted by VDL [4], is incorporated into the 'refinement calculus' [28], the main focus of [17]'s Chapter 3.

A *specification statement*

$$x : [p, P]$$

consists of a *frame x* of variables (a list containing all those that may change), *precondition, p,* a predicate whose free variables denote the initial state of the computation and which represents the states from which termination is certain, and a *postcondition, P,* a binary predicate in initial and final states which specifies the computation when it terminates. Enabledness is captured by feasibility: those initial states from which termination in a final state is possible

$$p[x_0/x] \Rightarrow \exists x \cdot P(x_0, x)$$

(the substitution of x_0 for x in the precondition is a technicality required by the decision to use x as free variable in p).

The semantics of a specification statement is given (see, for example, [28]) as a predicate transformer

$$\varepsilon.(x:[p,P]).q \;:=\; p \wedge (\forall x \cdot P \Rightarrow q)[x/x_0]\,,$$

and the ordering on specification statements is that inherited from $\mathcal{T}$. So, since ε is in fact surjective, specification statements are 'the same' as $\mathcal{T}$. Finally, having gained experience of unification and the benefits it affords, our path has returned to the context of [17]'s Chapter 3.

3.6 Chapter 3 revisited

It may now be appreciated that, from the viewpoint of unification, [17]'s Chapter 3 contains two unifications, performed almost effortlessly because they occur within the same, predicative, model [13]. To proceed in reverse order, a model of 'feasible specification statements' is defined as the subspace of single predicates satsifying all four healthiness conditions **H1** $\wedge$ **H2** $\wedge$ **H3** $\wedge$ **H4**. A model of 'not-necessarily-feasible specification statements' is defined by just the first three: **H1** $\wedge$ **H2** $\wedge$ **H3**. And the general space of *designs*, of use whenever enabledness, *ok*, and termination, ok', are observable, is defined by just the first two **H1** $\wedge$ **H2**.

Following the approach of the present paper, the original model of specification statements as predicate transformers [28] is adopted, and Galois connections are defined to relate to those other models. Chapter 3 is highly elegant in making those connections actually injections. It does so by using a predicative semantics with implication for refinement, using single-predicates (compared with the p and P in the previous section) and moreover by establishing an isomorphism between certain laws and healthiness conditions on the semantic space of predicates, which it then captured by closure operators. Little wonder, perhaps, that the reader may be distracted from the task of unification.

Indeed most of that is subservient to the primary concern of unification. What is the further benefit of ensuring that a model has that particular form? Of primary importance is unification of new paradigms of computation and the use of the unifying framework to simplify reasoning about realistic case studies. Surely that kind of endeavour is of secondary importance and may even look precious from outside the tight-knit UTP community.

That is why starting from Chapter 4 has been advocated, and only later returning to Chapter 3 to see the special nature of the relational/predicative injections.

4 Unifying further

The field of program semantics is specialised and any single approach to it, like UTP, even more so. Much of our hope for UTP must therefore lie in further applications of unification and the techniques UTP provides, outside the confines of program semantics. What for programming languages was *semantics* is now thought of as *behaviour*.

The best examples are the complex systems currently preoccupying us: hybrid systems like cyberphysical systems and those from biology and finance. Can the hierarchical approach be used to describe them incrementally in such a way that desirable properties 'accumulate'? That would make accessible 'closed form' analysis, to complement simulation and model checking which appear to be the sole techniques used at present.

The theories provided for incremental development, as summarised at the end of Section 2.2, are founded on a *uniform* domain $\mathcal{X}$ of discourse. A typical example is provided by the refinement calculus, which makes explicit the types of all variables appearing in the development. Thus when a development step involves a data refinement, both abstract and concrete spaces are included in $\mathcal{X}$. But data refinement is a special kind of increment which by definition prohibits observation of information encapsulated in the concrete data type, which is instead accessed only using the same operations as the abstract type.

In the setting of complex systems it may well be impractical to conceive the domain $\mathcal{X}$ *ab initio*. Instead, the complexity of the system may be revealed incrementally by successive Galois connections, following the approach of unification.

Here is an example from hardware design.

4.1 Beyond programming

The Boolean model of signal values provides a satisfactory account of hardware devices at one level of abstraction. Unfortunately it is quite abstract so, for realistic design, simulations (typically in HSPICE) based on lower-level models are required. One of the difficulties is in unifying the detailed model with the Boolean model. This seems like an ideal test for the UTP approach.

For example in the Boolean model, a wire connected to power by a p-type transistor is accurately modelled as being high if the gate of the transistor is low. But if the p-type transistor is replaced by an n-type transistor, the Boolean model predicts the same result, which is wrong: the wire is only weakly high, a result not able to be expressed in the model (but which is fatal because a chain of such transistors successively reduces the signal until it is not merely weakly high, but low).

A further observation—of 'drive'—needs to be incorporated in the model. This has been achieved elegantly by Hoare [16]. Each device is modelled first at the Boolean level (as is standard) but then at the driven level (this is new) and properties of the models ensure that the first is unified in the second. In fact both are embedded in predicates and the second extends the first, in the style of Chapter 3. Again, the situation is as in Figure 4, with the language being that of devices, $\mathcal{P}$ the Boolean model and $\mathcal{Q}$ the driven model.

The *Boolean model* is given by the set of predicates whose free variables are wire names from some set

$$
\begin{array}{rcl}
[\![pow]\!]_{\mathcal{B}} & := & out \\
[\![ntran]\!]_{\mathcal{B}} & := & g \Rightarrow (s = d) \\
[\![pow]\!]_{\mathcal{H}} & := & [\![pow]\!]_{\mathcal{B}} \wedge \delta out \\
 & = & out \wedge \delta out \\
[\![ntran]\!]_{\mathcal{H}} & := & \left(\begin{array}{l} [\![ntran]\!]_{\mathcal{B}} \\ g \wedge \delta g \wedge (\neg s \vee \neg d) \ \Rightarrow \ (\delta s = \delta d) \end{array} \right)
\end{array}
$$

Figure 10: Two devices, power and an n-type transistor, seen in two semantic models: the Boolean model $\mathcal{B}$ and the driven model $\mathcal{H}$.

say W and whose ordering is equivalence (since implication is too weak for the usual reason)

$$\mathcal{B}(W) \ := \ (pre.W, =)$$

For example the device *pow* which connects output *out* to power is modelled by the predicate $out = true$. An n-type transistor *ntran* with gate g, source s and drain d is modelled by the predicate which states that if the gate is high then source and drain equilibrate. See Figure 10.

In the 'driven model' an extra Boolean observable δw is included for each wire w in the Boolean model, representing whether or not that wire is driven to its value. For example the output of power is always driven and so its description in the driven model is its Boolean description conjoined with $\delta out = true$. The driven description of the n-type transistor consists of its Boolean description conjoined with a predicate relating drive of wires to their values: if the gate is driven high and either source or drain is low then when they equilibrate, as is guaranteed from the Boolean description, they are equi-driven. See Figure 10.

Thus the *driven model* extends the Boolean model by also containing a predicate whose free variables are both the wires and their δ version. Its order conjoins the Boolean order with the assurance that the driven predicate Δ' of the finer device is stronger than that, Δ, of the coarser:

$$\mathcal{H}(W) \ := \ (pre.W \times pre.(W \cup \delta W), \preceq)$$

where

$$(B, \Delta) \preceq (B', \Delta') \ := \ \left(\begin{array}{c} B = B' \\ \Delta' \Rightarrow \Delta \end{array} \right) .$$

Those examples suffice to confirm the example of weak signals mentioned above. But our concern here is with the unification. The Boolean model $\mathcal{B}$ is embedded in the driven model $\mathcal{H}$ by *injection*; and the ordering of $\mathcal{H}$ is stronger than that of $\mathcal{B}$. Thus the embedding is universally $\wedge$-junctive and the models are related by a Galois embedding.

Suppose it is required to model greater device detail. For example capacitance may be modelled as persistence of drive—say after a cycle's delay. That is captured by a third model, the capacitive model, in which the driven model is embedded. If, again, it is necessary to reason about time in more detail, a fourth model could be defined in which one cycle is replaced by a clock, so that a signal value and its single-cycle delay are replaced by a signal with values at discrete times. And so on. The state information required in more detailed models may be much more detailed than that of the abstract models (just Booleans, in this case), but nonetheless the relationship is mediated by Galois connections.

The case being made is that the techniques developed in UTP stretch far beyond theories of programming. They may be advantageously used to model, and reason about, complex systems.

4.2 The Philosopher's stone?

When will the UTP approach, of unification, not be helpful? When the incremental approach fails: when each feature is coupled so tightly with the others that the full behaviour cannot be 'teased out' into strands enabling it to be understood by approximation.

Consider a physical example. The *n*-body problem [6] requires the determination of the motion of *n* bodies, given their momenta at one instant and assuming Newtonian interactions. Specification of the problem is easy, by differential equation; the challenge lies in finding the solution. The problem is difficult because it must take into account all possible interactions between the bodies. There seems to be no scope for unification unless approximation is allowed. In Physics, approximation is a natural step to take because small changes in the momenta of the bodies lead to small changes in the solution. So one can imagine progressively more accurate solutions. In the case of discrete systems that kind of approximation is of little use (how do you approximate a bit?), and any method must instead approximate complexity *exactly* at each level of abstraction, through a series of abstractions. In that sense unification is our version of approximation in Physics. In the *n*-body it seems unachievable.

In the terms of Computer Science, the *n*-body problem is a distributed system in which each process interacts with each other. That, then, is going to be difficult to analyse incrementally unless there is some very special structure to the interactions. But if a process interacts with only a small number of others (for example its nearest neighbours, if they are distributed spatially) then unification might be expected.

5 Conclusion

Systems are inherently complicated. Since detail cannot ultimately be avoided, theories must be as simple as possible. In the areas of traditional engineering, where relationships between observables are assumed to be differentiable, approximation by simpler behaviours which approximate closely that of the real system, provides a successful method. It has been argued that unification, describing complex behaviour exactly at varying levels of abstraction, is the equivalent for the discrete systems of Computer Science.

In studying a complex system the first stage, then, must be to study its abstractions (ignore real time, the hybrid nature of the system and so on). But then must come a stage in which detail is restored. Then unification is our only technique. We conclude that every effort must therefore be made to sustain the theory of unification, UTP.

Unification might be appreciated as one of two 'orthogonal' techniques. That of *modularisation* structures descriptions at a given level of abstraction. *Unification* structures complexity incrementally across levels of abstraction. The former is reasonably well understood, is still being productively pursued at the research level (information hiding), and is the foundation of almost all Software Engineering. The UTP community appears to be guardians of the latter.

A single case study of the incremental approach has been presented, moving from predeterministic (*i.e.* computable) computations through finitely nondeterministic programs to angelic and arbitrary nondeterministic commands. The journey could readily have been continued to include probabilistic computations and even quantum computations (to go in just one direction). At most of the increments the semantic intuition and laws have been able to be lifted by Galois connections. Where that has not been the case, valuable insight has been provided by the property that fails (for example failure of *wp* to map intersections to disjunctions).

Though founded on unification, UTP offers further delightful distractions along the way. Many of them are compressed in to Chapter 3, and so the case has been made that, in teaching, attention be gently deflected to Chapter 4, then its predecessor viewed in context. Perhaps 'relational semantics' is not as important as might be thought from Chapter 3. As has just been seen in Section 3, it is not at all *required* for unification.

It has been suggested that unification offers a way of analysing complex systems, not just theories of programming. Indeed it has been claimed that only by diversifying from program semantics will the techniques of UTP be properly and widely appreciated. It would be very persuasive were the method to be used on complex systems currently being analysed by simulation or model checking, like hybrid systems arising from cyberphysical, biological or financial study. But within the confines of program semantics, it would be interesting to unify the standard models with more recent models, like the game theoretic model.

Many important topics have been overlooked in this paper. Just two are: the use of Galois connections for *calculation* by use of 'trading'; and data refinement in the domain of discourse and seen in terms of a Galois connection.

What, then, lies in store for UTP? It has been argued that the approach it takes, and the techniques it provides for unifying theories, are scientifically indispensible. But it has also been acknowledged that important ideas wither. Is UTP becoming a road less travelled, destined for obsolescence? The former appears to be true; the latter may be up to us. It seems obvious that (unless it is rediscovered) the approach will die without serious action: more courses might be taught, promoting unification; more students be engaged in MSc. and PhD. degrees based on UTP; more unification be performed, mastering new paradigms, making non-specialists want to use the method—good opportunities are provided by hybrid, cyberphysical and biological systems; and an undue amount of effort not be spent on second-order concerns. Otherwise, UTP will be as familiar in 20 years' time as are Zeppelins, Theremins and the slide rule.

What we call the beginning is often the end
And to make an end is to make a beginning.
The end is where we start from.

. . .

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.

Little Gidding [10]

References

[1] R.-J. R. Back and J. von Wright. Refinement calculus, Part I: Sequential nondeterministic programs. *LNCS*, **430**:42–66, Springer Verlag, 1989.

[2] R.-J. R. Back and J. von Wright. Duality in specification languages: a lattice-theoretical approach. *Acta Informatica*, **27**(7):583–625, 1990.

[3] J. W. Backus *et al.*, editors. Revised report on the algorithm language ALGOL 60. *CACM*, **6**(1):1–17, 1963. (Supplement by M. Woodger, 18–20.)

[4] D. Bjørner and C. B. Jones. The Vienna development method: The meta-language. *LNCS*, **61**, Springer Verlag, 1978.

[5] N. J. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[6] F. Diacu. The solution of the n-body problem. *The Mathematical Intelligencer*, **18**:66–70, 1996.

[7] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, **18**:453–457, 1975.

[8] R. Duke and G. Rose, *Formal Object-Oriented Specification Using Object-Z*, Macmillan Press, 2000.

[9] S. Dunne and W. Stoddart, editors. *Unifying Theories of Programming, First International Symposium, UTP 2006*, Walworth Castle, County Durham, UK, February 5-7, 2006, Revised Selected Papers, *LNCS*, **4010**, Springer Verlag, 2006.

[10] T. S. Eliot. *Four Quartets*. Harcourt, Inc., 1943.

[11] P. H. B. Gardiner, C. E. Martin and O. de Moor. An algebraic construction of predicate transformers. In *Mathematics of Program Construction*, *LNCS*, **669**:100-121, Springer Verlag, 1993.

[12] I. J. Hayes (editor). *Specification Case Studies*. Prentice-Hall International, 1987.

[13] E. C. R. Hehner. Predicative programming, parts I and II. *Communications of the ACM*, **27**(2):134–151, 1984.

[14] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, **2**(4):335–355, 1973.

[15] C. A. R. Hoare *et al*. The laws of programming. *Communications of the ACM*, **30**(8):672–686, 1987.

[16] C. A. R. Hoare. A theory for the derivation of combinational C-MOS circuit designs. *Theoretical Computer Science*, **90**(1):235–251, 1991.

[17] C. A. R. Hoare and He, Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[18] I. Houston and S. King. CICS project report: Experiences and results from the use of Z in IBM. *VDM Europe*, **1**:588–596, 1991.

[19] M. B. Josephs. Receptive process theory. *Acta Informatica*, **29**(1):17–31, 1992.

[20] H. Jerome Keisler. *Elementary Calculus: An Infinitesimal Approach*, second edition, Prindle, Weber and Schmidt, 1986. both by

[21] D. E. Knuth. The remaining troublespots in ALGOL 60. *Communications of the ACM*, **10**(10):611–617, 1967.

[22] A. M. Lister. *Fundamentals of Operating Systems*, second edition. Macmillan, 1979.

[23] P. Lucas and K. Walk. On the formal description of PL/I. In *Annual Review in Automatic Programming 6*, eds. M. I. Halpern and C. J. Shaw, 105–182. Pergamon Press, Oxford, 1971.

[24] A. K. McIver and C. C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer Verlag, 2005.

[25] R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.

[26] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[27] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[28] C. C. Morgan. *Programming from Specifications*, first edition. Prentice-Hall International, 1990.

[29] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, **9**(3):287–306, 1987.

[30] P. Nauer, editor. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, **6**:299–314, 1960.

[31] G. Nelson. A generalisation of Dijkstra's calculus. *ACM Transactions on Programming Language and Systems*, **11**(4):517–561, 1989.

[32] F. Nielsen, H. R. Nielsen and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 2005.

[33] I. M. Rewitzky. Binary Multirelations. In *Theory and Applications of Relational Structures as Knowledge Instruments, LNCS,* **2929**:256–271, Springer Verlag, 2003.

[34] I. M. Rewitzky and J. W. Sanders. Involutions on relational program calculi. *Scientific Annals of Computer Science*, **18**:129–171, 2008.

[35] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1998.

[36] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theoretical Computer Science*, **60**(2):177–229, 1988.

[37] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[38] J. M. Spivey. *The Z Notation: A Reference Manual*, second edition. Prentice-Hall International, 1992.

[39] A. van Wijngaarden *et al*. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, **5**:1–236, 1975.

[40] Wikipedia on the competition between VHS and Betamax:
`http://en.wikipedia.org/wiki/Videotape_format_war`.