Electronic Notes in Theoretical Computer Science 42 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume42.html 20 pages

# Abstract Commands: A Uniform Notation for Specifications and Implementations

Steve Dunne [1]

*School of Computing and Mathematics, University of Teesside*
*Middlesbrough, TS1 3BA, UK*

**Abstract**

Total correctness and general correctness are examined, and the latter is promoted as the more appropriate semantic basis for the programs which comprise the basic sequential interacting components of today's typically highly distributed systems, for which at certain times non-termination may actually be required. To this end we present a new abstract programming language, our abstract command language, defined with a full general-correctness semantics, and an appropriate notion of refinement which preserves general correctness and in respect of which all the constructors of our language are monotonic.

This provides a uniform notation for an abstract specification which expresses such a non-termination requirement and its stepwise and piecewise refinement into an executable implementation which meets such a requirement. Moreover, we illustrate the use of one of our language's constructors, *concert*, which enables such non-termination requirements to emerge during piecewise refinement, and we describe its implementation via a primitive form of concurrency.

## 1 Introduction

In 1996, while studying refinement in the B method, we encountered a class of programs, eventually to be called *semi-decidable* [2] programs, whose abstract specifications could be expressed neither in B's abstract machine notation nor even in its underlying generalised substitution language. Investigation soon established that the inexpressibility was not just a specific shortcoming in B's particular notations, but rather a fundamental limitation of the underpinning total-correctness semantics on which B's program derivation calculus was

[1] Email: s.e.dunne@tees.ac.uk

[2] A *decidable* program will always terminate with a correct result. A *semi-decidable* program will either terminate with a correct result or it won't terminate at all; i.e. it will never terminate, even abortively, with an incorrect result.

founded. It was therefore a limitation which B shared with the many other formal development methods which are, like B, founded on total correctness.

We first concentrated our attention on how the shortcoming might be remedied specifically within the context of B, and our initial proposals were published in [7]. Our ideas were further refined in [8] to provide a general-correctness counterpart of B's generalised substitutions, which we called *abstract commands*. The present paper consolidates that work by putting abstract commands on a formal footing independent of B to provide a uniform conceptual framework for expressing and reasoning about such semi-decidable programs at any level of abstraction.

In what follows we begin with a brief historical appreciation of predicate-transformer semantics leading up to the current dichotomy between the total-correctness and general-correctness schools. We argue that it is general correctness which offers us the fine grain of behavioural description which is needed by developers of the sequential programs which comprise the basic interacting building-blocks of today's increasingly distributed and intercomunicating systems. The second half of the paper is occupied with a detailed presentation of the syntax and semantics of abstract commands, in particular defining and discussing in turn each construct of the language.

## 2 Partial and Total Correctness

The celebrated weakest-precondition (wp) predicate-transformer semantics that Dijkstra [3] introduced for programs in 1976, and which was subsequently widely disseminated by Gries in [10], is a semantics of *total correctness*. It tells us under what starting conditions a program will be guaranteed to deliver a result satisfying a given postcondition. A less demanding semantics called *partial correctness* is characterised by a second predicate-transformer that Dijkstra also described but didn't use in [3]. He called this the weakest-liberal-precondition (wlp) predicate-transformer. It tells us under what starting conditions a program can be relied on to deliver a result satisfying a given postcondition providing it terminates at all –that is, when it can be relied on at least not to deliver a result contradicting the given postcondition.

Total correctness and partial correctness are obviously intimately linked. The difference between them is the issue of whether or not the program will terminate and so deliver any result at all. But partial correctness cannot simply be extracted from total correctness just by setting aside the question of termination. Nor can total correctness be reconstructed purely from partial correctness.

We will write $wp(A, Q)$ to represent the weakest precondition for a program $A$ to establish the postcondition $Q$, and in particular, therefore, $wp(A, \text{true})$ to represent the weakest precondition for $A$ to establish true, i.e. to be certain to terminate with any result at all. Correspondingly, we will write $wlp(A, Q)$ to represent the weakest precondition for $A$ to "liberally" establish

$Q$, i.e. establish $Q$ if it terminates at all. Although the predicate transformers $wlp(A, \_)$ and $wp(A, \_)$ are both positively conjunctive, which is to say they distribute across non-empty conjunctions, only $wlp(A, \_)$ is universally conjunctive, which is to say in addition it maps the empty conjunction, true, to itself. The necessary logical relationship between the two predicate-transformers is expressed by the law

$$wp(A, Q) \quad = \quad wp(A, \text{true}) \wedge wlp(A, Q) \ .$$

The termination predicate $wp(A, \text{true})$ of program $A$ is sometimes written as $trm(A)$. The above law intimates that wlp is more fundamental than wp. We could take $trm(A)$ and $wlp(A, \_)$ as the fundamental characteristics of $A$, and then simply define $wp(A, Q)$ as $trm(A) \wedge wlp(A, Q)$. Indeed, this is the approach we choose to adopt later in this paper in defining our abstract commands.

## 3 General Correctness

By the time [4] appeared in 1990, fourteen years after [3], it is clear Dijkstra had arrived at the view that a semantics which fully characterises the behaviour of sequential programs must embrace both total correctness *and* partial correctness. Thus [4] gives both a wp and a wlp interpretation of every construct in the guarded command language. Similarly, Nelson [22] employs both wp and wlp in his seminal generalisation of Dijkstra's calculus to admit abstract programs with miraculous behaviour and unbounded non-determinism. Nowadays the term *general correctness*, coined by Jacobs and Gries [18], denotes the semantics which combines both partial and total correctness.

However, by no means has everyone followed the lead of Dijkstra and Scholten, and Nelson, in adopting general correctness as the benchmark semantics for sequential programs. The most prominent schools of formal development, including B [1], VDM [19], Z [23] and the Refinement Calculus [21], all adhere to a notion of refinement based purely on total correctness. Surprisingly, even the recent work of Hoare and He [17,16] on unifying theories of programming ignores general correctness. Indeed, in [17] they promulgate a healthiness condition, H2, which explicitly forbids making non-termination an actual requirement, thereby ensuring that only designs characterised entirely by total correctness are admitted in their theory. In the next section we will argue why such an exclusive adherence to total correctness may be inappropriate for many of today's software systems.

Aside: the interesting question of how the Hoare and He unified theory itself might be tempered, without prejudicing its efficacy or elegance, to admit the wider class of predicates for general-correctness designs, we have already addressed in [5]. We do not pursue it here.

## 4 The Interactive Era

Milner has said that today computing *is* interaction.[3] This has important implications for software developers. For example, no longer are we always necessarily exclusively concerned with the result of executing our sequential programs through to successful termination. Such programs provide the individual interacting threads of today's complex distributed processing systems, and thus we may require them to behave reliably even in situations where termination is not even in prospect. A channel-listener, for example, must idle until it detects an incoming message, then deal effectively with such a message if and when it arrives. But since there is never a guarantee that such a message will ever arrive, our channel-listener must be capable in principle of idling indefinitely. There are circumstances, therefore, where non-termination *is* precisely what we demand. As formal software specifiers and designers we must have a specification language in which we can formally articulate such a requirement, and a design calculus with which we can prove that a given program meets such a specification. To quote from Hehner and Gravell [15]:

> With the addition of communication, non-terminating executions can perform useful computation, so a semantics that does not insist on termination is useful.

We would go further and say we need a semantics in which we can insist on non-termination in certain conditions. In short, we must learn to work in the context of general correctness.

In this regard it is also salient to point to work undertaken by various authors in the field of timed refinement, for example Hayes [11,12], and Hehner [13,14]. Such work invariably exhibits the crucial general-correctness characteristic that genuine non-termination is precisely characterisable. Total correctness, on the other hand, can only subsume non-termination into the completely unpredictable behaviour known as divergence or chaos. General correctness could be said implicitly to possess a very coarsely-grained domain for time with just three values: zero (for a computation's start), finite and positive (for a terminating computation's finish), and infinite (for a non-terminating computation's finish). Timed refinement might then be regarded as being founded on an elaborated variant of the general-correctness model in which time is accorded instead a much finer-grained continuum of values.

## 5 Refinement

In [21] Morgan advocates the discarding of any conceptual distinction between programs and specifications: all are programs, he says, though distributed across a wide refinement spectrum; some programs are more abstract while

[3] R. Milner, FACS $21^{st}$ anniversary symposium address, The Royal Society, London, 2 December 1998.

others are more concrete, and of the latter some are sufficiently concrete to be directly executable. Similarly, in [2] Back and Butler say

> In the refinement calculus, the required behaviour of the program is specified as an abstract, possibly non-executable, program which is then refined by a series of correctness-preserving transformations into an efficient, executable program. The notion of correctness-preserving transformation is modelled by a refinement relation between programs which is transitive, thus supporting stepwise refinement, and is monotonic with respect to the program constructors [4], thus supporting piecewise refinement.

We shall use the term *program* from now on in this abstract sense.

Morgan achieved his conceptual unification of programs and specifications by augmenting Dijkstra's guarded command language with his invention of the specification statement [20]. Naturally enough, since the semantics of guarded commands known to him at that time would have been that of total correctness, Morgan only endowed his specification statements with a total-correctness semantics. At about the same time Abrial was inventing generalised substitutions as the basis of the abstract machine notation in the B method of software development [1]. These can be used to describe the same range of abstract programs as specification statements, but in a more uniform notation that can largely be shared with concrete programs. This uniformity was Abrial's deliberate aim: in the B Method he sought to render the path from abstract specification to concrete code as seamless as possible. Like Morgan with his specification statements, Abrial gave his generalised substitutions only a total-correctness semantics. As we have already noted in the introduction, our abstract commands were originally inspired as the direct general-correctness counterparts of Abrial's generalised substitutions.

General-correctness refinement is a natural extension of total-correctness refinement. If $A$ and $B$ are generally-correct specifications over the same program space then $A$ is refined by $B$, written $A \sqsubseteq B$, if every possible behaviour of $B$ is a possible behaviour of $A$ too. Formally, this is expressed by asserting that for every postcondition $Q$ over the program space

$$wp(A, Q) \Rightarrow wp(B, Q) \qquad \text{and} \qquad wlp(A, Q) \Rightarrow wlp(B, Q) \ .$$

General-correctness programs form a complete lattice under this refinement relation.

Dijkstra and Scholten's own re-interpretation in [4] of the guarded command language in terms of general correctness, didn't add any capability for abstract specification. On the other hand, Nelson's generalisation [22] of the guarded command language did give it both new ingredients: an expressivity encompassing abstract programs, and a general-correctness semantics. It

[4] They mean, of course, rather that the program constructors are monotonic with respect to the refinement relation!

is unfortunate then, that Nelson's creation falls foul of Back and Butler's monotonicity stipulation quoted above; its program constructors are not all monotonic with respect to refinement. The offending ones are if ... fi and do ... od, because from $A \sqsubseteq B$ we can infer neither

$$\text{if } A \text{ fi} \quad \sqsubseteq \quad \text{if } B \text{ fi} \qquad \text{nor} \qquad \text{do } A \text{ od} \quad \sqsubseteq \quad \text{do } B \text{ od} \; .$$

Thus Nelson's generalisation of the guarded command language fails to provide us with a practical medium for program development, since it doesn't support piecewise refinement.

## 6 Iteration in General Correctness

When we move from total correctness into general correctness the refinement ordering on programs is no longer appropriate as a basis for the standard least-fixed-point treatment of iteration as a special case of recursion. It would, for example, equate the infinite loop program

while true do *skip* end

of our abstract command language with the completely chaotic program we call *anarchy* which is our refinement bottom. But, operationally speaking at least, our infinite loop is completely deterministic since it can never terminate. General correctness must faithfully reflect its guaranteed non-termination. The standard way to achieve this, as Nelson explains in [22], is to use another ordering and define recursions as least fixed-points with respect to this instead. The ordering in question is the so-called Egli-Milner approximation ordering: when $A$ and $B$ are programs, we say $A$ *approximates* $B$ if for every postcondition $Q$

$$wp(A, Q) \Rightarrow wp(B, Q) \qquad \text{and} \qquad wlp(B, Q) \Rightarrow wlp(A, Q) \; .$$

Although this looks at first glance very similar to our earlier refinement ordering, note the crucial distinction that the two implications are now in opposite directions. It turns out that our infinite loop is the bottom of the Egli-Milner ordering since it approximates every program, whereas the refinement bottom *anarchy* approximates only the small class of programs with no constraint at all on their results whenever they do terminate.

In fact later we will define abstract command iteration constructively using natural induction, as did Dijkstra originally in [3] for the do ... od of his guarded command language. Thus we avoid any appeal to the Egli-Milner ordering and fixed-point theory. Nevertheless, to introduce a more complete notion of recursion into our abstract command language we would have to employ Egli-Milner.

## 7 Abstract Commands

The notion of an active frame was first introduced by us in [6] for generalised substitutions. We now adopt it for abstract commands too. We call a collection of variables a *frame*. If a frame comprises only one variable, then the frame and its single constituent variable are synonymous. If $u$ and $v$ are frames, then $u \cup v$ denotes the new frame obtained by merging $u$ and $v$, and $u \setminus v$ denotes the residual frame obtained by removing from $u$ any variables it shares with $v$. We say the frame $v$ *extends* the frame $u$ if $u \subseteq v$. We denote the empty frame by $\emptyset$ .

An abstract command is always interpreted in a context provided by a set of variables which forms the global reference frame or *alphabet* of all variables whose names are in scope. If we ever have to make explicit reference to it as a frame, we denote the alphabet by $\alpha$. We assume, without explicitly defining it, an implicit total ordering over the alphabet which allows us to use frames as vector variables in an unambiguous way.

We call the collection within the alphabet of variables on which an abstract command acts its *frame*. Our operational intuition is that these are the variables to which the command may assign values. We denote the frame of an abstract command $A$ by $\mathit{frame}(A)$. An abstract command may make passive reference to variables in the alphabet outside its frame. For example, the frame of $x := y$ is just $x$ although it makes passive reference to $y$ too. The frame may be empty as in *skip*, or as in $x < 7 \mid \mathit{skip}$ which makes only passive reference to $x$. We distinguish between *skip* and $x := x$ since they have different frames.

We can now proceed to define the basic commands and constructors constituting our abstract command language. We list them all in Table 1 for ease of reference. The meaning of an abstract command $A$ comprises three distinct elements: its frame $\mathit{frame}(A)$, its termination $\mathit{trm}(A)$ and its weakest-liberal-precondition predicate-transformer $\mathit{wlp}(A, \_)$ which acts on postconditions, i.e. predicates over the current alphabet. Accompanying the following definitions will often be an operational interpretation of the command concerned. Such interpretations are provided only to assist our intuition about the command. They are never a necessary part of the command's definition, which is always given formally in terms of the three above elements.

**skip**: as one can infer from its name *skip* does nothing, and is therefore easy to characterise: $\mathit{frame}(\mathit{skip})$ is the empty frame, $\mathit{trm}(\mathit{skip})$ is always true, and $\mathit{wlp}(\mathit{skip}, \_)$ is the identity predicate transformer. So we have

$$\begin{aligned} \mathit{frame}(\mathit{skip}) &= \emptyset \,, \\ \mathit{trm}(\mathit{skip}) &= \text{true} \,, \\ \mathit{wlp}(\mathit{skip}, Q) &= Q \,. \end{aligned}$$

| name | syntax |
|---|---|
| skip | skip |
| assignment | $x := E$ |
| preconditioned command | $P \mid A$ |
| guarded command | $P \Longrightarrow A$ |
| bounded choice | $A[]B$ |
| conditional | if $P$ then $A$ else $B$ end |
| short conditional | if $P$ then $A$ end |
| unbounded choice | $@\ z\ .\ A$ |
| sequential composition | $A\ ;B$ |
| finite repetition | $A^n$ |
| indeterminate assignment | $x : P$ |
| repetitive closure | $A^*$ |
| iteration | while $P$ do $A$ end |
| parallel composition | $A \mid\mid B$ |
| concert | $A\#B$ |

Table 1
The Abstract Command Language

**assignment**: takes the form $x := E$ where $E$ is a well-defined expression of appropriate type. The frame of $x := E$ is $x$ and it always terminates. The predicate-transformer $wlp(x := E, \_)$ is defined in terms of the syntactic substitution $Q\langle E/x\rangle$, which denotes $Q$ with every free occurrence of $x$ replaced by $E$. So we have

$$\begin{aligned} frame(x := E) &= x\ , \\ trm(x := E) &= \text{true}\ , \\ wlp(x := E, Q) &= Q\langle E/x\rangle\ . \end{aligned}$$

**preconditioned command**: takes the form $P \mid A$ where $P$ is a predicate and $A$ is an abstract command. The precondition $P$ simply has the effect of strengthening $A$'s termination predicate without affecting its frame or wlp.

We therefore have

$$\begin{aligned} \mathit{frame}(P \mid A) &= \mathit{frame}(A) \ , \\ \mathit{trm}(P \mid A) &= P \wedge \mathit{trm}(A) \ , \\ \mathit{wlp}(P \mid A, Q) &= \mathit{wlp}(A, Q) \ . \end{aligned}$$

**guarded command**: takes the form $P \Longrightarrow A$ where $P$ is a predicate and $A$ is an abstract command. The guard $P$ restricts $A$'s feasibility. We have

$$\begin{aligned} \mathit{frame}(P \Longrightarrow A) &= \mathit{frame}(A) \ , \\ \mathit{trm}(P \Longrightarrow A) &= P \Rightarrow \mathit{trm}(A) \ , \\ \mathit{wlp}(P \Longrightarrow A, Q) &= P \Rightarrow \mathit{wlp}(A, Q) \ . \end{aligned}$$

In particular, the form $P \Longrightarrow \mathit{skip}$ is the manifestation in our abstract command language of what is historically known as a Floyd assumption [9].

The interaction between preconditions and guards in our general-correctness semantics gives us important expressive power. By preconditioning a guarded command, we can express guaranteed non-termination. For example, we define a program *never* by

$$\mathit{never} = \text{false} \mid \text{false} \Longrightarrow \mathit{skip} \ .$$

We can easily show from this definition that $\mathit{trm}(\mathit{never}) = \text{false}$ and, for any Q, $\mathit{wlp}(\mathit{never}, Q) = \text{true}$ , which means that *never* is never guaranteed to terminate, but will give us any result we ask for if it does terminate (even if we ask the impossible, a result satisfying false). We conclude that *never* can, in fact, never terminate. Its definition is an extreme example of the interesting form $P \mid P \Longrightarrow A$, which prescribes behaviour like $A$ where $P$ holds and guaranteed non-termination outside $P$. This corresponds to the new general-correctness meaning given to if $P \longrightarrow A$ fi in the guarded command language by Dijkstra and Scholten [4], and Nelson [22]. And if we make $A$ *skip* again, the astute reader will this time recognise $P \mid P \Longrightarrow \mathit{skip}$ as a Floyd assertion [9].

**bounded choice**: takes the form $A \,[]\, B$ where $A$ and $B$ are abstract commands. It represents a demonic choice between $A$ and $B$. Their frames are merged, so we have

$$\begin{aligned} \mathit{frame}(A \,[]\, B) &= \mathit{frame}(A) \cup \mathit{frame}(B) \ , \\ \mathit{trm}(A \,[]\, B) &= \mathit{trm}(A) \wedge \mathit{trm}(B) \ , \\ \mathit{wlp}(A \,[]\, B, Q) &= \mathit{wlp}(A, Q) \wedge \mathit{wlp}(B, Q) \ . \end{aligned}$$

**conditional**: takes the familiar form if $G$ then $A$ else $B$ end where $G$ is a predicate and $A$ and $B$ are abstract commands. It is defined by

$$\text{if } G \text{ then } A \text{ else } B \text{ end} \quad = \quad (G \Longrightarrow A) \,[]\, (\neg\, G \Longrightarrow B) \ .$$

A similar definition is given by Abrial in [1] in the context of generalised substitutions. As Abrial has pointed out, such a definition is interesting because it reveals that what was long regarded by programmers as a fundamental programming construct is in fact merely a compounding of the two more fundamental constructs of guarding and non-deterministic choice.

**short conditional**: takes the familiar form if $G$ then $A$ end where $G$ is a predicate and $A$ is an abstract command. It is defined by

$$\text{if } G \text{ then } A \text{ end} \quad = \quad (G \Longrightarrow A) \,[]\, (\neg\, G \Longrightarrow \mathit{skip}) \ .$$

**unbounded choice**: takes the form $@z \,.\, A$ where $A$ is an abstract command, and $z$ is fresh with respect to the current alphabet. The alphabet of $A$ is understood as the current alphabet augmented by $z$. The unbounded choice $@z \,.\, A$ represents $A$ attenuated by a demonic choice of any value for variable $z$. The frame of $@z \,.\, A$ is obtained by removing $z$, should it be there, from $A$'s frame. We therefore have

$$\begin{aligned}
\mathit{frame}(@z \,.\, A) &= \mathit{frame}(A) \setminus z \ , \\
\mathit{trm}(@z \,.\, A) &= \forall\, z \bullet \mathit{trm}(A) \ , \\
\mathit{wlp}(@z \,.\, A, Q) &= \forall\, z \bullet \mathit{wlp}(A, Q) \ .
\end{aligned}$$

The construct earns its name from the fact that if $z$ ranges over an infinite domain of values the choice will be unboundedly non-deterministic. Thus it may break Dijkstra's *Law of Continuity* [3].

**sequential composition**: takes the form $A \,;\, B$, where $A$ and $B$ are abstract commands. It represents execution of $A$ followed by $B$. The frames of $A$ and $B$ are merged, and $A \,;\, B$ terminates where $A$ terminates and is guaranteed to establish $B$'s termination. So we have

$$\begin{aligned}
\mathit{frame}(A \,;\, B) &= \mathit{frame}(A) \cup \mathit{frame}(B) \ , \\
\mathit{trm}(A \,;\, B) &= \mathit{trm}(A) \wedge \mathit{wlp}(A, \mathit{trm}(B)) \ , \\
\mathit{wlp}(A \,;\, B, Q) &= \mathit{wlp}(A, \mathit{wlp}(B, Q)) \ .
\end{aligned}$$

**finite repetition**: takes the form $A^n$ where $A$ is an abstract command and $n$ is a natural number. It represents $n$ successive executions of $A$. We define $A^n$ inductively in terms of sequential composition as follows:

$$\begin{aligned}
A^0 &= \mathit{skip} \ , \\
A^{n+1} &= A \,;\, A^n \ .
\end{aligned}$$

## 8 Soundness

Now that we have defined some specific abstract commands, it is appropriate to raise the question of the soundness of these definitions. In general, when defining a new abstract command $A$, do we have to respect any constraints on $\mathit{frame}(A)$, $\mathit{wlp}(A, \_)$ and $\mathit{trm}(A)$, or are we free to define each of them arbitrarily?

In fact, we can specify $\mathit{trm}(A)$ arbitrarily as any predicate over the alphabet, but, as we have already noted in section 2, $\mathit{wlp}(A, \_)$ must be universally conjunctive. Furthermore, there is a mutual constraint called frame consistency between $\mathit{frame}(A)$ and $\mathit{wlp}(A, \_)$ which ensures that $A$ cannot exert an inappropriate effect on variables outside its frame: if $Q$ is a postcondition which is independent of $\mathit{frame}(A)$ in that no variable of $\mathit{frame}(A)$ is free in $Q$, then $\mathit{wlp}(A, Q)$ must be an attenuation, or weakening, of $Q$: in other words, $Q \Rightarrow \mathit{wlp}(A, Q)$ must hold. For many commands the attenuation will be trivial in that $Q = \mathit{wlp}(A, Q)$ holds, but strict weakening arises when $A$ is infeasible. For example, $\mathit{wlp}(\text{false} \Longrightarrow \mathit{skip}\ , \_)$ attenuates any $Q$ completely to true. Any abstract command $A$ expressible through the abstract command language constructors defined in this paper necessarily satisfies

(i) $\mathit{wlp}(A, \_)$ distributes across arbitrary conjunctions, and

(ii) $Q \Rightarrow \mathit{wlp}(A, Q)$ holds for any $Q$ independent of $\mathit{frame}(A)$.

## 9 Abstract Command Refinement

We now raise the important question of what it means to say one command refines another. Earlier, we introduced the notion of general-correctness refinement, but we only defined it then for programs over the same program-space. That gives us a suitable notion of refinement for abstract commands sharing the same frame, yet we need more than this if our abstract command language is to furnish a complete program development environment in Back and Butler's sense. A comprehensive notion of refinement over all abstract commands on an alphabet must take frames into account. For example, can we refine *skip* by $x := x$, or indeed $x := x$ by *skip*? In fact, the extra ingredient for abstract command refinement concerns frame inclusion. Thus, if $A$ and $B$ are abstract commands over the same alphabet, then $A$ is refined by $B$, written $A \sqsubseteq B$, if and only if, for every postcondition $Q$ over the alphabet,

$$\mathit{trm}(A) \Rightarrow \mathit{trm}(B)\ ,$$

$$\mathit{wlp}(A, Q) \Rightarrow \mathit{wlp}(B, Q)\ ,\ \text{and}$$

$$\mathit{frame}(A) \subseteq \mathit{frame}(B)\ .$$

Under our abstract command refinement *skip* is indeed refined by $x := x$ but the converse is not so. All the abstract command language constructors

defined in this paper are monotonic with respect to our abstract command refinement. The frame extension aspect of abstract command refinemnt ensures the parallel composition we will define in section 12 is monotonic.

## 10 Characteristic Predicates

We have already encountered one fundamental predicate of an abstract command $A$: its termination predicate, $trm(A)$, which characterises from where execution of $A$ is guaranteed to terminate, and is therefore safe from the risk of non-termination. Several other important characteristic predicates are associated with $A$. The names of most of them are adopted like *trm* from [1] where they are used for corresponding predicates defined for generalised substitutions.

**fis**(**A**): the feasibility predicate $fis(A)$ characterises from where execution of $A$ is feasible (non-miraculous). Remember, a program behaves miraculously where it can guarantee to establish false, so we define it by

$$fis(A) \quad = \quad \neg\, wp(A, \text{false})$$

or, equivalently,

$$fis(A) \quad = \quad \neg\, trm(A) \lor \neg\, wlp(A, \text{false}) \ .$$

**cic**(**A**): the perpetuity predicate $cic(A)$ characterises from where perpetual repetition of the abstract command $A$ is feasible. (The name *cic* is an acronym for "cycles and infinite chains".) It is defined by

$$cic(A) \quad = \quad \forall\, n : Nat \bullet fis(A^n) \ .$$

**saf**(**A**): the safety predicate $saf(A)$ characterises from where any finite repetition of $A$ is safe from the risk of non-termination. We define it by

$$saf(A) \quad = \quad \forall\, n : Nat \bullet trm(A^n) \ .$$

**prd**(**A**): the before-after predicate $prd(A)$ relates before-values of the variables in $A$'s frame to their potential after-values following execution of $A$. Let $frame(A)$ be represented by $x$; then we define $prd(A)$ by

$$prd(A) \quad = \quad \neg\, wlp(A, x \neq x') \ .$$

where $x'$ is fresh with respect to the current alphabet and represents the after-value of $x$. The predicate $x \neq x'$ is called the before-after discrepancy. For a frame comprising variables $x, y$ the before-after discrepancy would be $x, y \neq x', y'$ which expands to $\neg\, (x = x' \land y = y')$. In the case of an empty

frame the conjunction degenerates to true, so the before-after discrepancy associated with an empty frame is false. Hence, for example,

$$prd(skip) \quad = \quad \neg\, wlp(skip, \text{false}) \quad = \quad \neg\, \text{false} \quad = \quad \text{true} \; .$$

If $frame(A)$ is already known then $prd(A)$ and $wlp(A, \_)$ carry equivalent information about $A$, so that each is derivable from the other. For example, if $frame(A)$ is $x$ we have that

$$wlp(A, Q) \quad = \quad \forall\, x' \bullet prd(A) \Rightarrow Q\langle x'/x \rangle \; .$$

This is useful since sometimes when we wish to introduce a new abstract command $A$ it is more convenient to specify $prd(A)$ than $wlp(A, \_)$.

## 11 Of Demons and Miracles

The interplay between guarded commands and demonic choice which we exploited to decompose the traditional conditional, relies, so to speak, on the demon's innate abhorrence of miracles. Paradoxically, demonic choice is angelic with respect to feasibility. This is illustrated very starkly when we involve the everywhere-miraculous abstract command $magic$ ( $=$ false $\Longrightarrow$ $skip$) in a demonic choice with any abstract command $A$. It is easy to show that

$$A \;[]\; magic \quad = \quad A \; .$$

The demon's freedom of choice is constrained by feasibility. When offered a choice between a miraculous and a feasible alternative, he must always take the latter.

Indeed, the demon is more constrained than might appear from the immediate choice confronting him: offered a choice between two apparently feasible immediate alternatives, it may be the case that one of those choices leads to an infeasible command later, perhaps much later, in the program. The demon must recoil from even such deferred infeasibility. For example, it is easy to show that the program

$$(x := 1 \;[]\; x := 2) \;;\; x = 1 \Longrightarrow skip$$

is equivalent to $x := 1$. Nelson [22] suggests we adopt either of two equivalent operational intuitions about the demon's behaviour. We could attribute to him a clairvoyant ability to look into the future course of execution of the program to foresee any infeasibility that would result from taking a particular choice, so as to avoid that choice. Alternatively, we can imagine that he makes his choice blindly but that execution will backtrack to let him reconsider if infeasibility is subsequently encountered. In the little program above we observe that the $x = 1 \Longrightarrow skip$ acts as a retrospective "choice filter" to ensure the demon makes the choice we want in the preceding command.

## 12 Further Abstract Commands

**indeterminate assignment**: takes the form $x : P$ where $x$ is a variable in the current alphabet and $P$ is a predicate over the current alphabet plus $x'$. It represents the assignment to $x$ of any value of $x'$ satisfying $P$. We have

$$\begin{aligned} \mathit{frame}(x : P) &= x \ , \\ \mathit{trm}(x : P) &= \text{true} \ , \\ \mathit{prd}(x : P) &= P \ . \end{aligned}$$

Indeterminate assignment isn't a fundamental abstract command. We can equivalently define it by

$$x : P \ = \ @x'.\ P \Longrightarrow x := x' \ .$$

We can express any abstract command $A$, where $\mathit{frame}(A) = x$, in the form

$$\mathit{trm}(A) \mid x : \mathit{prd}(A) \ ,$$

which provides a useful normal form for abstract commands. It also provides a convenient way of defining our refinement bottom command *anarchy* by

$$\mathit{anarchy} \ = \ \text{false} \mid \alpha : \text{true} \ .$$

This is one of the very few cases where we have to make explicit reference to $\alpha$, our alphabet frame.

**repetitive closure**: takes the form $A^*$ where $A$ is an abstract command. It represents an arbitrary repetition of $A$: that is, the demonic choice of any finite repetition or even perpetual repetition of $A$. We have

$$\begin{aligned} \mathit{frame}(A^*) &= \mathit{frame}(A) \ , \\ \mathit{trm}(A^*) &= \mathit{saf}(A) \ \wedge \ \neg \ \mathit{cic}(A) \ , \\ \mathit{wlp}(A^*, Q) &= \forall \, n : \mathit{Nat} \bullet \mathit{wlp}(A^n, Q) \ . \end{aligned}$$

The definition of $\mathit{trm}(A^*)$ can be understood in the light of the fact that there are two distinct risks of non-termination here: first, any particular execution of $A$ might not terminate; second, the repetition itself might be perpetual, which would also manifest as non-termination. We are safe from the first risk where $A$ is safe, and from the second where perpetual repetition of $A$ isn't feasible: i.e. where $\mathit{cic}(A)$ is false. In fact, $A^*$ can be shown to be the Egli-Milner least fixed-point $X$ of the abstract command expression

$$(A \ ; X) \ [] \ \mathit{skip} \ .$$

It is fundamental in our definition of iteration which follows.

**iteration**: takes the familiar form while $G$ do $A$ end where $G$ is a predicate and $A$ is an abstract command. It is defined by

$$\text{while } G \text{ do } A \text{ end} \quad = \quad (G \Longrightarrow A)^* \; ; \neg \; G \Longrightarrow \mathit{skip} \; .$$

Credit for such an ingenious deconstruction of what had always hitherto been regarded as a fundamental programming construct must go to Abrial, who similarly decomposes B's WHILE DO END in [1]. But whereas we have used our constructively defined abstract command repetitive closure, Abrial employs what he calls the *transitive opening* of a generalised substitution, in the definition of which he has to appeal to fixed-point theory.

The $\neg \; G \Longrightarrow \mathit{skip}$ acts as a choice filter: it constrains the demon executing the repetitive closure preceding it to choose exactly whichever finite repetition of $G \Longrightarrow A$ (there can only be one, if it exists at all) is feasible and makes $G$ false. Conversely, if every finite repetition of $G \Longrightarrow A$ leaves $G$ true then the demon is obliged to choose everlasting repetition, should this be feasible, so the iteration doesn't terminate. Of course, if everlasting repetition is infeasible –and this can arise, even though $G$ holds, through $A$'s own inherent infeasibility– then the demon will be utterly confounded by lack of choice; the whole iteration is in that case infeasible. It is easy to prove from the above definition that

$$\text{while true do } \mathit{skip} \text{ end} \quad = \quad \mathit{never}$$

which vindicates our earlier operational interpretation of *never* as an abstract specification of an infinite loop.

**parallel composition**: takes the form $A \mid\mid B$ where $A$ and $B$ are abstract commands. Execution proceeds until $A$ and $B$ have both terminated, and the result, if any, is the composite effect of $A$ and $B$. Here, rather than specify $\mathit{wlp}(A \mid\mid B, \_)$ directly, it is convenient to specify $\mathit{prd}(A \mid\mid B)$. As usual the frames are merged, so we have

$$\begin{aligned} \mathit{frame}(A \mid\mid B) \quad &= \quad \mathit{frame}(A) \cup \mathit{frame}(B) \; , \\ \mathit{trm}(A \mid\mid B) \quad &= \quad \mathit{trm}(A) \wedge \mathit{trm}(B) \; , \\ \mathit{prd}(A \mid\mid B) \quad &= \quad \mathit{prd}(A) \wedge \mathit{prd}(B) \; . \end{aligned}$$

Our parallel composition is quite general: it doesn't even require the frames of $A$ and $B$ to be disjoint. If their frames are disjoint it represents independent executions of $A$ and $B$. A reviewer of the draft version of this paper expressed some misgiving about the above definition of $\mathit{trm}(A \mid\mid B)$ since, as he said, "in general parallel composition with overlapping frames, the termination of the whole is not a simple function of the termination of the parts". But our parallel operator is in fact no more than a particular case of [17]'s *parallel by merge* where, to quote Hoare and He,

> Each process is first executed on its *private* version of the shared variables independently. When all have terminated, their updates on the shared variables are *merged* and written back to the *global* version of the shared variables.

In our case this merge is the straightforward feasible fusion [2] of the two sets of results. Where the respective effects of $A$ and $B$ on a common variable are wholly irreconcilable $A \parallel B$ is infeasible. We defined a similar parallel composition in [6] for generalised substitutions. Our definition here is actually simpler than that one, since total correctness induces a dependency between the *trm* and *prd* of a generalised substitution which is not mirrored in general correctness, where *trm* and *prd* are independent of each other.

**concert**: takes the form $A\#B$, where $A$ and $B$ are abstract commands. It represents parallel execution of $A$ and $B$ on disjoint copies of their respective frames in a termination pact. These concerted executions proceed until either terminates. The overall result, if any, is determined entirely by whichever has terminated. Their frames are merged, so we have

$$\begin{aligned} \mathit{frame}(A\#B) &= \mathit{frame}(A) \cup \mathit{frame}(B) \ , \\ \mathit{trm}(A\#B) &= \mathit{trm}(A) \vee \mathit{trm}(B) \ , \\ \mathit{wlp}(A\#B, Q) &= \mathit{wlp}(A, Q) \wedge \mathit{wlp}(B, Q) \ . \end{aligned}$$

We might imagine concert being concretely implemented in, say, a Unix/C environment by the parent process forking two subsidiary processes, then waiting for either to terminate, upon which it just kills off the other one. Thus concert offers us the means of refining a decidable specification into a concurrently executing pair of semi-decidable programs. We illustrate this in the next section with a domestic allegory.

## 13 A Lost Ring

Suppose we have lost a valuable ring at home, either in the garden or in the house. Let $A$ be the specification "Find the lost ring". Let $P$ be the predicate "The ring is in the garden", and therefore $\neg\, P$ be "The ring is in the house". Then

$$P \Longrightarrow A$$

is the specification "Find the lost ring which is (assumed to be) in the garden", and if we then precondition this by $P$, we obtain

$$P \mid P \Longrightarrow A$$

which is equivalent to the semi-decidable specification "Search the garden for the lost ring", semi-decidable because if the ring is in the garden this

process will find it, but if not it prescribes a never-terminating garden search. Correspondingly,

$$\neg\ P \mid \neg\ P \Longrightarrow A$$

is equivalent to the semi-decidable specification "Search the house for the lost ring". If we have a gardener who is familiar with the garden and a housekeeper who is familiar with the house these two searches are easy to implement separately. But we note that they must be executed in parallel. We know our housekeeper and our gardener will each be indefatigable in their respective searches and never admit defeat. It may well be futile therefore, for example, first to ask the housekeeper to undertake her house-search, and await its outcome before deciding whether or not to ask the gardener to undertake his garden-search. For there may never be any outcome to the housekeeper's search, since as long as she should fail to find the ring she will just keep on searching. We can prove that for any abstract command $A$ and any predicate $P$

$$A \ \ = \ \ (P \mid P \Longrightarrow A) \ \ \# \ \ (\neg\ P \mid \neg\ P \Longrightarrow A)$$

which demonstrates the efficacy of our concurrent search strategy as a means of implementing our overall search requirement. Indeed, this may be the only feasible implementation if there is no-one familiar with the intimate geography of both house and garden.

## 14 Monotonicity of Concert

Above all the others, perhaps, concert is our quintessential abstract command constructor. But, one might reasonably ask, is it really as special as all that? After all, couldn't we with a little ingenuity have defined an analgous operator in the total-correctness world of generalised substitutions? For example, what about

$$trm(S) \vee trm(T) \ \mid \ (trm(S) \Longrightarrow S \ [] \ trm(T) \Longrightarrow T) \ \ ?$$

Doesn't this express the behaviour of generalised substitutions $S$ and $T$ in concert? The answer, perhaps surprisingly, is yes: It does express a substitution which will always act like whichever of $S$ and $T$ is guaranteed to terminate whenever one of them is. This certainly seems to express the idea of two concurrent programs co-operating so that whichever of them is guaranteed to finish defines the result of the overall computation. So then, why can't this constructor be used to divide a complicated computation into two simpler ones, to be implemented separately and then executed concurrently in such a termination pact? The answer is simply that this constructor isn't monotonic with respect to total-correctness refinement. Just like Nelson's do ... od, therefore, it is useless for piecewise program development.

In contrast, concert enjoys, like all our other abstract command constructs, the essential property of being monotonic with respect to general-correctness refinement. Thus it can be employed in piecewise program development. This is the crucial point about general correctness. Within its context we can define *usable* new operators such as concert, which even if they are definable in total correctness are ultimately unimplementable there.

## 15 Conclusion

We have compared the total-correctness and general-correctness treatments of program semantics, arguing that the latter is an appropriate foundation for many of today's computing needs with their accent on interactivity. We have presented our abstract command language, derived from Abrial's generalised substitution language, but appropriately founded on a semantics of general correctness.

Since in the B method a generalised substitution is invariably interpreted within the context of an abstract machine whose state variables are expected provide the effective frame context, Abrial's generalised substitution language itself is equipped with only a weak implicit notion of frame. A significant feature of our abstract command language is its explicit and wholly compositional treatment of frames, in the spirit of Morgan's specification statements [21]. This facilitates a precise and rigorous description of the frame-expanding or frame-contracting effect of the various constructors of the language.

All the constructors of our abstract command language are monotonic with respect to abstract-command refinement, so the language provides a uniform self-contained system for stepwise and piecewise algorithmic development in general correctness of an abstract specification into an executable implementation. We are aware of no other specification language which provides a comparable context for such development; our abstract command language therefore, we believe, is an original contribution which will facilitate the expression both of an abstract specification in general correctness, and of its subsequent refinement towards implementation.

## References

[1] Abrial, J.-R., "The B-Book: Assigning Programs to Meanings," Cambridge University Press, 1996.

[2] Back, R. and M. Butler, *Fusion and simultaneous execution in the refinement calculus*, Acta Informatica **35** (1998), pp. 921–940.

[3] Dijkstra, E., "A Discipline of Programming," Prentice-Hall International, 1976.

[4] Dijkstra, E. and C. Scholten, "Predicate Calculus and Program Semantics," Springer, Berlin, 1990.

[5] Dunne, S., *Recasting the Hoare-He unified theory of programming in the context of general correctness* Technical Report. School of Computing and Mathematics, University of Teesside, 2000.

[6] Dunne, S., *The Safe Machine: a new specification construct for B*, in: J. Wing, J. Woodcock and J. Davies, editors, *FM'99 - Formal Methods*, number 1708 in Lecture Notes in Computer Science (1999), pp. 472–489.

[7] Dunne, S., W. Stoddart and A. Galloway, *Extending the generalised substitution to model semi-decidable operations*, in: H. Habrias, editor, *The First B Conference* (1996), iSBN 2-906082-25-2.

[8] Dunne, S., W. Stoddart and A. Galloway, *Specification and refinement in general correctness*, in: A. Evans, D. Duke and A. Clark, editors, *Proceedings of the 3rd Northern Formal Methods Workshop* (1998), http://www.ewic.org.uk/ewic/workshop/view.cfm/NFM-98.

[9] Floyd, R., *Assigning meanings to programs*, Proceedings of Symposia in Applied Mathematics **19** (1967), pp. 19–32.

[10] Gries, D., "The Science of Programming," Springer-Verlag, 1981.

[11] Hayes, I., *Separating timing and calculation in real-time refinement*, in: J. Grundy, M. Schwenke and T. Vickers, editors, *International Refinement Workshop and Formal Methods Pacific 1998* (1998), pp. 1–16.

[12] Hayes, I., *Reasoning about non-terminating loops using deadline commands*, in: R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC2000)*, 2000, also available as Technical Report UQ-SVRC-00-02, http://svrc.it.uq.edu.au.

[13] Hehner, E., *Termination is timing*, in: J. van de Snepscheut, editor, *Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science (1989), pp. 36–47.

[14] Hehner, E., "A Practical Theory of Programming," Springer-Verlag, 1993.

[15] Hehner, E. and A. Gravell, *Refinement semantics and loop rules*, in: J. Wing, J. Woodcock and J. Davies, editors, *FM'99 - Formal Methods*, number 1709 in Lecture Notes in Computer Science (1999), pp. 1497–1510.

[16] Hoare, C., *Theories of programming: Top-down and bottom-up and meeting in the middle*, in: J. Wing, J. Woodcock and J. Davies, editors, *FM'99 - Formal Methods*, number 1708 in Lecture Notes in Computer Science (1999), pp. 1–27.

[17] Hoare, C. and H. Jifeng, "Unifying Theories of Programming," Prentice Hall, 1998.

[18] Jacobs, D. and D. Gries, *General correctness: a unification of partial and total correctness*, Acta Informatica **22** (1985), pp. 67–83.

[19] Jones, C., "Systematic Software Development Using VDM (2nd edn)," Prentice-Hall, 1990.

[20] Morgan, C., *The specification statement*, ACM Transactions on Programming Languages and Systems **10** (1988).

[21] Morgan, C., "Programming from Specifications (2nd edn)," Prentice Hall International, 1994.

[22] Nelson, G., *A generalisation of Dijkstra's calculus*, ACM Transactions on Programminig Languages and Systems **11** (1989).

[23] Spivey, J., "The Z Notation: a Reference Manual (2nd edn)," Prentice Hall International, 1992.