Acta Informatica 25, 1–14 (1988)

Termination Conventions and Comparative Semantics

Eric C.R. Hehner and Andrew J. Malton
University of Toronto, CSRI, Sandford Fleming Building, 10 King's College Road, Toronto, Canada, M5S 1A4

Summary. The notion of termination is examined, first for its physical observability, then for its part in six semantic formalisms, with emphasis on predicative semantics.

Observing Termination

Imagine there is a computer in front of you right now. Perhaps it has a screen and keyboard, as is common. You can press the keys, and watch the patterns on the screen. Unfortunately, this computer was built and programmed by Tatooinians, whose language you do not know. The symbols on the keys are meaningless to you; the patterns on the screen are undecipherable. Fortunately, no-one is asking you what is being computed, or what the computation means. You are being asked a simpler question: When has the computation terminated? If you will simply report when the machine is finished, someone else will then interpret the results.

How do you know when termination occurs? If nothing has happened for five minutes, can you report termination? No; perhaps the machine is computing *Ackermann* (6,6), and still has a century to go. It is obvious that nontermination, or infinite looping, is not an observable event; without knowing the program, you can only guess that a computation is stuck in a loop. The purpose of this thought-experiment is to convince you that termination is also not an observable event.

How do you recognize termination on your own computer, programmed by someone who speaks your language? Probably you look for the word "done", or something similar, printed as the dying act. Or perhaps you look for a prompt from the operating system, following termination. It is possible, though not probable, that your computer has broken in mid-computation, and is now behaving erratically; it just printed the letters "done" (or the prompt) by chance as it malfunctions. Or perhaps the program was written by a joker, who thinks it funny to print "done" (or the prompt) in the middle of the computation, followed in a minute by "wait, there's more". You cannot be certain when a computation has finished even on your own machine, although the usual indicators are very reliable.

Predicative Semantics

A predicative formalism is a formalism for specifying and describing computations, including the semantics of programs, as predicates. One is invited to decide what observable quantities are of interest, and to express specifications as predicates having these quantities as free variables. A computation satisfies a specification if it instantiates the free variables in a way that satisfies the predicate.

A reasonable choice of "observable quantities of interest" is the communication sequence, or sequences, to and from a computer. In many computer systems and programming languages, there is something that indicates termination of a communication sequence. (In UNIX it is control-D; in Pascal it is the *eof* function.) But at a basic semantic level, this end-of-file indication is just another communication. It is only by agreement that we give it the special meaning "don't wait for more". We may decide to enshrine this meaning in a programming language's semantics; there are some advantages in doing so. Similarly, we can design into a language a communication with the meaning "please enter input now", and many other equally useful messages. Or instead, we may decide that the programmers, not the language designers, give meaning to the communications. We leave it to them to decide upon whether and how to prompt, to decide whether and how to say "this message is the last". The benefit is a simpler semantics.

A recent paper [3] presented and used a predicative formalism in which the observable quantities of interest are taken to be the initial values $\grave{x}$, $\grave{y}$, ... and final values $\acute{x}$, $\acute{y}$, ... of some variables. Pointedly, termination and nontermination are not taken to be observable. It is left to the programmers to decide upon their termination convention. We assume knowledge of [3], or of its forerunner [2], and we now discuss the possible termination conventions.

Stability Conventions

A termination convention, or more generally a *stability convention*, is a predicate $\sigma(v)$ of the program variables (or state) v that is not identically true:

$$\neg\forall v \cdot \sigma.$$

A state satisfying σ is called "stable", and a state not satisfying σ is called "unstable". (We disallow the trivial predicate "true" as a stability convention to avoid having to say "non-trivial stability convention" in the theorems and comparative semantics that follow.) Here are two examples.

Convention u (unstable state). Let us call one of the states u. Then, under this convention, we define $\sigma = (v \neq u)$. In words, u is taken to be an unstable state, and all other states are considered stable. (The symbol $\perp$ is sometimes used for a fictitious state reached after nontermination. In contrast, we intend the unstable state u to be an actual machine state; it is the state during a computation, after its start and before its termination. We are not considering u to be the bottom of a lattice.)

Convention b (busy bit). Let us call one boolean variable b. Then, under this convention, we define $\sigma = \neg b$. We might implement b as a light on the console. Before the computation has started, it is off (false); during the computation, it is on (true); when it goes off (false), the state is again stable and the computation has terminated.

From a coding viewpoint, convention u is the most economical; it wastes only one state to indicate instability, leaving all others to indicate results. Convention b makes half the states unstable; still, it wastes only one bit. From a recognition viewpoint, convention b is the most practical; we need to look at only one bit, not all the bits, to determine stability.

Respectfulness

In [3] we defined the "determined" operator ∇ for any specification $S(\grave{v}, \acute{v})$ as

$$\nabla S = \neg \forall \acute{v} \cdot S.$$

∇S is true of exactly those inputs $\grave{v}$ for which we are interested in the output $\acute{v}$. We also considered a special case of the invariant construct

$$(\mathbf{inv}\, P \cdot S) = (\grave{P} \wedge \nabla S \Rightarrow S \wedge \acute{P})$$

where P is any predicate of the program state. In these terms, we now define "respectfulness".

A specification S is said to *respect* convention σ iff

$$\forall \grave{v} \cdot \forall \acute{v} \cdot S = (\mathbf{inv}\, \sigma \cdot S).$$

The following six theorems afford us some insight into respectfulness.

Theorem 0. *S respects σ iff*

$$\forall \grave{v} \cdot \nabla S = \grave{\sigma} \wedge (\forall \acute{v} \cdot S \Rightarrow \acute{\sigma}).$$

For a determined specification, the intent is that the initial state should be stable, and any final state satisfying S should be stable. During the computation, the machine is in an unstable state. Suppose for a moment that we can observe the state during a computation. Under convention u, since there is only one unstable state, we cannot see *how* the computation is progressing – only *that* it is. Under convention b, we may be able to watch the sequence of unstable states, and see how the machine works. But our specifications do not speak of intermediate, unstable states; in that sense, they are not "of interest".

Theorem 1. *Specification $S(\grave{v}, \acute{v})$ respects convention u iff it is both strict*

$$S(u, u)$$

and explosive

$$\forall \grave{v}\cdot(S(\grave{v}, u) \Rightarrow \forall \acute{v}\cdot S(\grave{v}, \acute{v})).$$

This means that from initial state u, final state u is possible; and if, from some initial state, final state u is possible, then, from that initial state, every final state is possible.

Theorem 2. *Specification S respects convention b iff it is both strict*

$$\forall \grave{v}\cdot\exists \acute{v}\cdot \grave{b} \Rightarrow S \wedge \acute{b}$$

and explosive

$$\forall \grave{v}\cdot(\exists \acute{v}\cdot S \wedge \acute{b}) \Rightarrow \forall \acute{v}\cdot S.$$

Again, "strictness" means that instability may lead to instability, and "explosiveness" means that if S allows the final state to be unstable, then it allows every final state.

Theorem 3. *Let S be an arbitrary specification. Then* **inv** $\sigma\cdot S$ *respects convention* σ.

All our programming connectives that were defined in [3] have the property that if their parts respect some convention, then the whole does.

Theorem 4. *If P and Q respect σ, then so do $(P \circ Q)$, $(P; Q)$, (P* **or** *Q) and* (**if** *b* **then** *P* **else** *Q). If, assuming P respects σ, $S(P)$ does also, then* **loop** *P: $S(P)$ respects* σ.

It is especially nice that, under any stability convention, the difference between relational product and composition disappears.

Theorem 5. *If Q respects some convention σ, then*

$$P; Q = P \circ Q.$$

The Need for a Stability Convention

We must be able to decide when a computation is finished. Since termination is not an observable event, it is necessary to have and to respect a stability convention. But it is really of no interest which convention we choose. We would like to be able to write a simple specification that refers only to the desired results, not to the arbitrary convention. As a trivial example, we would like to write

$$\acute{x} = \grave{x} + 1$$

to say that x is increased by 1. Unfortunately, this specification does not respect any convention. To obtain a respectful specification, we apply Theorem 3. Under convention b we obtain

$$\neg \grave{b} \Rightarrow \acute{x} = \grave{x} + 1 \wedge \neg \acute{b}$$

and under convention u we obtain

$$\grave{v} \neq u \Rightarrow \acute{x} = \grave{x} + 1$$

assuming that the consequent somehow implies $\acute{v} \neq u$. Even the simple specification $\mathbf{ok} = (\acute{v} = \grave{v})$ does not respect any convention; we must instead write either $(\neg \grave{b} \Rightarrow \acute{v} = \grave{v})$ or $(\grave{v} \neq u \Rightarrow \acute{v} = \grave{v})$. All specifications become a little more complicated, with extra detail that is of no programming interest.

A stability convention is also necessary for the implementation of semi-colon: to implement $P; Q$ it must be possible to recognize termination of P's execution in order to start Q's execution. The usual convention used for that purpose involves part of the state (the "program counter") that programmers would prefer not to think about. No "higher-level" version of this implementation detail (such as convention u or convention b) is an improvement. Accordingly, semi-colon was defined in [3]

$$P; Q = (\nabla P \Rightarrow P \circ Q)$$

with an antecedent saying only that there can be a stability convention, but not specifying any particular one. This is the strongest predicate that can be implemented as sequential execution (for proof, see [3]).

Although a stability convention is of no interest to a programmer (in fact, it is an impediment), it allows us to translate among various semantic formalisms, and compare them. That is our purpose in the remainder of this paper. In these translations and comparisons, we shall refer to the predicative semantic formalism as PS.

Dijkstra's wp

According to Dijkstra's "weakest precondition" formalism [1], a program S is defined by a predicate transformer $\text{wp}(S, R)$ that maps any postcondition R to the corresponding necessary and sufficient precondition P. Its interpretation is that, starting in a state satisfying P, execution of S will terminate in a state satisfying R.

Suppose we have a PS specification S and a postcondition R, and we want to find precondition $\text{wp}(S, R)$. First, extend the state space with at least one unstable state to provide a stability convention σ (defined to be true of exactly the unextended state space). We assume that S respects σ (if not, replace S with $(\mathbf{inv}\, \sigma \cdot S)$ in the following). We also assume $(\forall v \cdot R \Rightarrow \sigma)$ (if not, replace R with $R \wedge \sigma$ in the following). Now

$$\grave{\text{wp}}(S, R) = (\forall \acute{v} \cdot S \Rightarrow \acute{R}).$$

(The accent on wp signifies only that the result is a predicate in $\grave{v}$; it may be deleted.) The result of this translation is a precondition P such that $(\forall v \cdot P \Rightarrow \sigma)$. The stability convention has now served its purpose; if desired, it can be thrown away by taking σ to be true, thus eliminating the unstable states.

For the reverse translation we are given $\text{wp}(S, R)$ for arbitrary R, and we want to express S as a predicate. It is

$$S = \mathbf{inv}\ \sigma \cdot \neg\, \grave{w}\mathrm{p}(S,\ \acute{v} \neq v).$$

(For the calculation of wp, the $\acute{v}$ are constants; the variables are unaccented. In the resulting precondition, place ` accents on the unaccented variables.) Again, we can throw away σ by letting it be true.

With the stability convention, translations between PS and wp do not lose information; they are reversible. Translation between PS and wp can also be made directly without introducing a stability convention. The translation is

$$\grave{w}\mathrm{p}(S, R) = \nabla S \wedge (\forall\, \acute{v} \cdot S \Rightarrow \acute{R})$$
$$S = \neg\, \grave{w}\mathrm{p}(S,\ \acute{v} \neq v).$$

But in one case (called **havoc** in the catalogue of semantics, later) this is not a reversible translation.

Jones's VDM

In Jones's VDM (Vienna Development Method) formalism [4], a specification is a pair $(\grave{P}, R)$ in which $\grave{P}$ is a precondition (predicate on the initial state) and R is a relation (predicate on the initial and final states). Its interpretation is that, starting in a state $\grave{v}$ satisfying $\grave{P}$, execution will terminate in a state $\acute{v}$ such that the pair $(\grave{v}, \acute{v})$ satisfies R.

To translate, first extend the state space with at least one unstable state to provide a stability convention σ. Then, given PS specification S respecting σ, we obtain VDM precondition $\grave{P}$ and relation R as follows:

$$\grave{P} = \nabla S, \qquad R = S.$$

In VDM, two specifications are considered equivalent if their preconditions are equivalent and their relations agree whenever $\grave{v}$ satisfies the precondition. So we could choose any relation R such that

$$(\nabla S \wedge S) \Rightarrow R \Rightarrow (\nabla S \Rightarrow S).$$

We shall consistently choose R to be as weak as possible.

In the reverse direction, given VDM specification $(\grave{P}, R)$ respecting σ, i.e. such that $(\forall\, v \cdot P \Rightarrow \sigma)$ and $(\forall\, \grave{v} \cdot \forall\, \acute{v} \cdot \grave{P} \wedge R \Rightarrow \acute{\sigma})$, obtain PS specification S as

$$S = (\grave{P} \Rightarrow R).$$

Once again, with the stability convention the translations do not lose information. The translation can be made without any stability convention, but then there is the same single case in which the translation is not reversible.

Translation between VDM and wp is reversible with or without a stability convention. Given wp,

$$\grave{P} = \grave{\mathrm{w}}\mathrm{p}(S, \mathbf{true}), \qquad R = \neg\, \grave{\mathrm{w}}\mathrm{p}(S, \acute{v} \neq v).$$

The other way, given $\grave{P}$ and R,

$$\grave{\mathrm{w}}\mathrm{p}(S, Q) = \grave{P} \wedge (\forall \acute{v} \cdot R \Rightarrow \acute{Q}).$$

Partial Relation PR

In this formalism, a specification consists of a (possibly partial) relation R, with the following interpretation: if $(\grave{v}, \acute{v})$ satisfies R, then execution starting at $\grave{v}$ must terminate, and $\acute{v}$ is a possible final state. This formalism has been proposed by Robison [6] and independently by B. von Stengel (private communication).

An implementable PS specification S can be translated to PR by extending the state space as before, and ensuring that S is respectful. Then

$$R = \nabla S \wedge S.$$

The reverse translation is

$$S = ((\exists \acute{v} \cdot R) \Rightarrow R).$$

Parnas's LD and SS

In Parnas's LD (Limited Domain) formalism [5], as in Jones's VDM, a specification is a pair $(\grave{C}, R)$. $\grave{C}$ is a predicate on the initial state called the "competence", and R is a relation (predicate on the initial and final states). Its interpretation is a little different than that of VDM: if execution starts in a state $\grave{v}$ satisfying $\grave{C}$, it will terminate in a state $\acute{v}$ such that the pair $(\grave{v}, \acute{v})$ satisfies R; if it starts in a state $\grave{v}$ not satisfying $\grave{C}$, it will either terminate in a state $\acute{v}$ such that the pair $(\grave{v}, \acute{v})$ satisfies R or it will fail to terminate. Unlike VDM, specifications that differ only when $\grave{v}$ lies outside the competence set are not considered equivalent.

The translation between PS and LD is similar to the translation between PS and VDM for specifications that respect a stability convention. LD differs from VDM in its ability to be disrespectful: an LD specification can say that, for some initial state, nontermination and termination in a limited set of final states are acceptable, but termination in any other final state is unacceptable.

$$\grave{C} = (\forall \acute{v} \cdot S \Rightarrow \acute{\sigma}), \qquad R = S \wedge \acute{\sigma}$$
$$S = (\grave{C} \vee \acute{\sigma} \Rightarrow R).$$

Parnas has also considered a relational semantics that he has called "standard semantics" SS. It is similar to LD but the competence is determinable from the relation as

$$\grave{C} = \exists \acute{v} \cdot R$$

and so we can dispense with it. SS can be seen as a variation of PR. In PR, if for $\grave{v}$ there is no $\acute{v}$ such that $(\grave{v}, \acute{v})$ satisfies R, then execution starting at $\grave{v}$ is arbitrary; in SS, nontermination is mandatory.

$$\begin{aligned} R = {} & \text{if } \forall \grave{v} \cdot (\forall \acute{v} \cdot S \Rightarrow \acute{\sigma}) \vee (\forall \acute{v} \cdot S \Rightarrow \neg \acute{\sigma}) \\ & \quad \text{then } S \wedge \acute{\sigma} \\ & \quad \text{else inexpressible} \\ S = {} & (\acute{\sigma} \Rightarrow (\exists \acute{v} \cdot R)) \wedge ((\exists \acute{v} \cdot R) \Rightarrow R). \end{aligned}$$

Catalogue of Semantics

We now present a dozen specifications and connectives, each in the six formalisms we have just described. Those that are basic (not composed from others) are shown both with and without a stability convention. The reader should compare them for simplicity and ease of expression. Our comparative comments follow.

specification called **abort**, **chaos** or **disaster**: arbitrary behavior

	with σ	*without* σ
PS	**true**	**true**
wp(S, R)	**false**	**false**
VDM	**false, true**	**false, true**
PR	**false**	**false**
LD	**false, true**	**false, true**
SS	inexpressible	inexpressible

specification called **havoc** or **chance**: terminating but otherwise arbitrary behavior

	with σ	*without* σ
PS	$\grave{\sigma} \Rightarrow \acute{\sigma}$	inexpressible
wp(S, R)	$\forall v \cdot \sigma \Rightarrow R$	$\forall v \cdot R$
VDM	$\grave{\sigma}$, $\grave{\sigma} \Rightarrow \acute{\sigma}$	**true, true**
PR	$\grave{\sigma} \wedge \acute{\sigma}$	**true**
LD	$\grave{\sigma}$, $\grave{\sigma} \Rightarrow \acute{\sigma}$	**true, true**
SS	inexpressible	**true**

specification called **skip**, **ok** or **continue**

	with σ	*without* σ
PS	$\grave{\sigma} \Rightarrow \acute{v} = \grave{v}$	$\acute{v} = \grave{v}$
wp(S, R)	$\sigma \wedge R$	R
VDM	$\grave{\sigma}$, $\grave{\sigma} \Rightarrow \acute{v} = \grave{v}$	**true**, $\acute{v} = \grave{v}$
PR	$\grave{\sigma} \wedge \acute{v} = \grave{v}$	$\acute{v} = \grave{v}$
LD	$\grave{\sigma}$, $\grave{\sigma} \Rightarrow \acute{v} = \grave{v}$	**true**, $\acute{v} = \grave{v}$
SS	inexpressible	$\acute{v} = \grave{v}$

specification called **miracle**

	with σ	*without* σ
PS	$\neg\grave{\sigma}$	**false**
wp(S, R)	σ	**true**
VDM	$\grave{\sigma}, \neg\grave{\sigma}$	**true, false**
PR	inexpressible	inexpressible
LD	$\grave{\sigma}, \neg\grave{\sigma}$	**true, false**
SS	inexpressible	inexpressible

assignment $x := e$ in two variables x, y, assuming e is evaluable

	with σ	*without* σ
PS	$\grave{\sigma} \Rightarrow \acute{x} = \grave{e} \wedge \acute{y} = \grave{y} \wedge \acute{\sigma}$	$\acute{x} = \grave{e} \wedge \acute{y} = \grave{y}$
wp(S, R)	$\sigma \wedge R[x\colon e]$	$R[x\colon e]$
VDM	$\grave{\sigma}, \grave{\sigma} \Rightarrow \acute{x} = \grave{e} \wedge \acute{y} = \grave{y} \wedge \acute{\sigma}$	**true**, $\acute{x} = \grave{e} \wedge \acute{y} = \grave{y}$
PR	$\grave{\sigma} \wedge \acute{x} = \grave{e} \wedge \acute{y} = \grave{y} \wedge \acute{\sigma}$	$\acute{x} = \grave{e} \wedge \acute{y} = \grave{y}$
LD	$\grave{\sigma}, \grave{\sigma} \Rightarrow \acute{x} = \grave{e} \wedge \acute{y} = \grave{y} \wedge \acute{\sigma}$	**true**, $\acute{x} = \grave{e} \wedge \acute{y} = \grave{y}$
SS	inexpressible	$\acute{x} = \grave{e} \wedge \acute{y} = \grave{y}$

sequential composition: satisfy first P then Q

PS	$\nabla P \Rightarrow P \circ Q$
wp(S, R)	wp$(P$, wp$(Q, R))$
VDM	$\grave{P}_P \wedge (\forall \acute{v} \cdot R_P \Rightarrow \acute{P}_Q)$, $R_P \circ R_Q$
PR	$(\forall v \cdot P(\grave{v}, v) \Rightarrow \exists \acute{v} \cdot Q(v, \acute{v})) \wedge P \circ Q$
LD	$\grave{C}_P \wedge (\forall \acute{v} \cdot R_P \Rightarrow \acute{C}_Q)$, $R_P \circ R_Q$

SS	if $\forall \grave{v} \cdot (\exists \acute{v} \cdot P \circ Q) \Rightarrow (\forall v \cdot P(\grave{v}, v) \Rightarrow \exists \acute{v} \cdot Q(v, \acute{v}))$
	then $P \circ Q$
	else inexpressible

deterministic choice: if b is initially true then satisfy P else satisfy Q

PS	$\grave{b} \wedge P \vee \neg\grave{b} \wedge Q$
wp(S, R)	$\grave{b} \wedge$ wp$(P, R) \vee \neg\grave{b} \wedge$ wp(Q, R)
VDM	$\grave{b} \wedge \grave{P}_P \vee \neg\grave{b} \wedge \grave{P}_Q$, $\grave{b} \wedge R_P \vee \neg\grave{b} \wedge R_Q$
PR	$\grave{b} \wedge P \vee \neg\grave{b} \wedge Q$
LD	$\grave{b} \wedge \grave{C}_P \vee \neg\grave{b} \wedge \grave{C}_Q$, $\grave{b} \wedge R_P \vee \neg\grave{b} \wedge R_Q$
SS	$\grave{b} \wedge P \vee \neg\grave{b} \wedge Q$

nondeterministic choice: satisfy either P or Q

PS	$P \vee Q$
wp(S, R)	wp$(P, R) \wedge$ wp(Q, R)
VDM	$\grave{P}_P \wedge \grave{P}_Q$, $\grave{P}_P \wedge \grave{P}_Q \Rightarrow R_P \vee R_Q$
PR	$(\exists \acute{v} \cdot P) \wedge (\exists \acute{v} \cdot Q) \wedge (P \vee Q)$
LD	$\grave{C}_P \wedge \grave{C}_Q$, $R_P \vee R_Q$

SS	if $\forall \grave{v} \cdot (\exists \acute{v} \cdot P) = (\exists \acute{v} \cdot Q)$
	then $P \vee Q$
	else inexpressible

joint satisfaction: satisfy both P and Q

PS	$P \wedge Q$
wp(S, R)	$\mathrm{wp}(P, R) \vee \mathrm{wp}(Q, R)$
VDM	$\grave{P}_P \vee \grave{P}_Q,\ R_P \wedge R_Q$
PR	if $\forall \grave{v} \cdot (\exists \acute{v} \cdot P \wedge Q) \vee (\neg \exists \acute{v} \cdot P) \vee (\neg \exists \acute{v} \cdot Q)$
	then $((\exists \acute{v} \cdot P \wedge Q) \Rightarrow P \wedge Q)$
	$\wedge ((\neg \exists \acute{v} \cdot P) \vee (\neg \exists \acute{v} \cdot Q) \Rightarrow P \vee Q)$
	else inexpressible
LD	$\grave{C}_P \vee \grave{C}_Q,\ R_P \wedge R_Q$
SS	if $\forall \grave{v} \cdot (\exists \acute{v} \cdot P \wedge Q) \vee (\neg \exists \acute{v} \cdot P) \wedge (\neg \exists \acute{v} \cdot Q)$
	then $P \wedge Q$
	else inexpressible

ordering: P satisfied if Q satisfied

PS	$P \Leftarrow Q$
wp(S, R)	$\forall R \cdot \mathrm{wp}(P, R) \Rightarrow \mathrm{wp}(Q, R)$
VDM	$\grave{P}_P \Rightarrow \grave{P}_Q \wedge (R_P \Leftarrow R_Q)$
PR	$(\exists \acute{v} \cdot P) \Rightarrow (\exists \acute{v} \cdot Q) \wedge (P \Leftarrow Q)$
LD	$\grave{C}_P \Rightarrow \grave{C}_Q \wedge (R_P \Leftarrow R_Q)$
SS	$((\exists \acute{v} \cdot P) = (\exists \acute{v} \cdot Q)) \wedge (P \Leftarrow Q)$

variable declaration: introduce fresh local variable x into P

PS	$\exists \grave{x} \cdot \exists \acute{x} \cdot P$
wp(S, R)	$\forall x \cdot \mathrm{wp}(P, R)$
VDM	$\forall \grave{x} \cdot \grave{P}_P,\ \exists \grave{x} \cdot \exists \acute{x} \cdot R_P$
PR	$\exists \grave{x} \cdot \exists \acute{x} \cdot P$
LD	$\forall \grave{x} \cdot \grave{C}_P,\ \exists \grave{x} \cdot \exists \acute{x} \cdot R_P$
SS	if $\forall \grave{v} \cdot \forall \acute{v} \cdot (\exists \grave{x} \cdot \exists \acute{x} \cdot P) \Rightarrow (\forall \grave{x} \cdot \exists \acute{x} \cdot P)$
	then $\exists \grave{x} \cdot \exists \acute{x} \cdot P$
	else inexpressible

recursion: to satisfy **loop** P: $S(P)$, satisfy $S($**loop** P: $S(P))$

PS	$\forall i \cdot S^i(\mathit{true})$
wp(S, R)	$\exists i \cdot \mathrm{wp}(S^i(\mathbf{abort}), R)$
VDM	$\exists i \cdot \grave{P}_{S^i(\mathbf{abort})},\ \forall i \cdot R_{S^i(\mathbf{abort})}$
PR	not easily expressible
LD	not easily expressible
SS	inexpressible.

Comparative Comments

It is questionable whether a semantic formalism that talks only about the initial input state and the final output state, without any interactive input or output during the computation, can be considered adequate. But all the formalisms considered here are of that sort, so it is not a point of comparison.

All of the formalisms provide a way of saying, for each input state, "specification P requires termination". They are:

PS	∇P
wp	wp(P, **true**)
VDM	$\grave{P}_P$
PR	$\exists \acute{v} \cdot P$
LD	$\grave{C}_P$
SS	$\exists \acute{v} \cdot P$.

The negation means, of course, "specification P does not require termination". Only two of the formalisms provide a way of saying "specification P requires nontermination". They are:

LD	$\neg \grave{C}_P \wedge \neg \exists \acute{v} \cdot P$
SS	$\neg \exists \acute{v} \cdot P$.

In PS, ∇P says that P is determined; to know that a computation satisfies a determined specification, one must observe its final state (so it must have one). But to know that a computation satisfies an undetermined specification, i.e. when $\forall \acute{v} \cdot P$, it is not necessary to observe its final state (so it does not need to have one).

One of the formalisms, wp, is not based on a relation. This makes it awkward to state that the output is to be related to the input. To say that x is to be increased by 1, we may write something like

$$\mathrm{wp}(S, x = X + 1)$$

accompanied by words that say X is the initial value of x. But formally this allows the unintended solution

$$X := 3; x := 4$$

for S. To say what is intended, we must be able to distinguish, in the formalism, between initial and final values.

VDM and PS are very close, as seen by the simplicity of the translation. In practice, it often makes sense to think of the precondition separately from the relation, as in VDM. One writes them down with a comma between, as in VDM, or with an implication sign, as in PS. VDM offers us a freedom in stating the relation, and we chose to make it as weak as possible. Consequently,

$$\forall \grave{v} \cdot \nabla R \Rightarrow \grave{P}.$$

This means that the precondition is entirely superfluous whenever ∇R is true. It gives us further information only for those inputs where ∇R is false. If, for some input, all outputs are acceptable, the precondition tells us whether termination is required. Its only service is to distinguish, for each input state, **chaos** from **havoc**. In judging the value of this service against its cost, keep in mind the fact that termination, by itself, is not observable.

LD differs from VDM in its ability to be disrespectful: an LD specification can say that, for some initial state, nontermination and termination in a limited set of final states are acceptable, but termination in any other final state is unacceptable. Such a specification is very easy to implement: ignore the final states and deliver a nonterminating loop. Hence the ability to be disrespectful is of no practical value in programming. PS, wp, VDM, and PR all hold that (without communications or intermediate results) infinite loops are useless, and there is no point in being particular about possible final states while allowing an infinite loop. One may take the position that LD can describe more mechanisms than PS, wp, VDM, or PR: it can describe those that can nondeterministically either terminate in a particular final state or not terminate. Or, one may instead take the position that mechanisms are deterministic, and that nondeterminism is a property of specifications. In that case, LD is no more descriptive than the other formalisms.

To a specifier, the most important connective is probably joint satisfaction. It is very common to specify by parts, stating properties that must be jointly satisfied. A specifier will therefore appreciate the formalism that makes joint satisfaction simplest.

To a programmer, the most important connective is undoubtedly the ordering, because that is the criterion for correctness of a program. Indeed, the entire programming process is to transform a specification, little by little, to a program, and at each stage to ensure that the new is properly related to the old by the ordering. A programmer will therefore appreciate the formalism that makes the ordering simplest.

Of all these formalisms, PR fits the class of computations best – in fact, perfectly. All implementable specifications are expressible, and all unimplementable specifications are not. That was its design criterion. To achieve this, PR significantly complicates the expression of its connectives, particularly joint satisfaction.

Alone among the formalisms, PS can be criticized for the fact that its semi-colon is not always associative. The counter-example creates, using semi-colon, what would be **havoc** in another formalism. For example, in one integer variable x,

$$(\acute{x} ⧧ \grave{x};\ \acute{x} ⧧ \grave{x});\ \acute{x}=0=\mathbf{true}$$

$$\acute{x} ⧧ \grave{x};\ (\acute{x} ⧧ \grave{x};\ \acute{x}=0)=\acute{x}=0.$$

Our intuition that semi-colon should be associative is simply this: executing first P and then $(Q; R)$ should be the same as executing first $(P; Q)$ and then R. This makes sense when execution of P and $(P; Q)$ terminate. But if P's execution fails to terminate, what does "and then $(Q; R)$" mean? If $(P; Q)$'s execution fails to terminate, what does "and then R" mean? It is a theorem of PS that semi-colon is associative under all circumstances called for by intuition:

$$\nabla P \wedge \nabla(P; Q) \Rightarrow (P; (Q; R) = (P; Q); R).$$

In fact, the stronger theorem

$$\nabla(P; Q) \Rightarrow \nabla P \wedge (P; (Q; R) = (P; Q); R)$$

holds.

More Power

We now present one more semantic formalism, called PF (Powerful Formalism). As in five of the formalisms presented so far, the heart of PF is a relation between input $\grave{v}$ and output $\acute{v}$. Like wp, VDM and LD, it has further information to impart. In PF a specification is a sextuple (ẁg, ẁlg, ẁp, ẁlp, R, śt) in which the first four components are predicates on the initial state, R is the relation, and śt is a predicate on the final state. These components are to be interpreted as follows.

ẁg:	true of an initial state from which the computation is required to progress, false of an initial state in which the computation is allowed to get stuck.
ẁlg:	true of an initial state from which the computation is allowed to progress, false of an initial state in which the computation is required to get stuck.
ẁp:	true of an initial state for which the computation is required to terminate, false of an initial state for which the computation is allowed to run forever.
ẁlp:	true of an initial state for which the computation is allowed to terminate, false of an initial state for which the computation is required to run forever.
R:	this relation describes the allowed transitions from initial to final state.
śt:	true of a final state iff termination is successful.

Here is a small sample of basic specifications in PF

chaos	= **false,**	**true,**	**false,**	**true,**	**true,**	**false**
havoc	= **true,**	**true,**	**true,**	**true,**	**true,**	**false**
random	= **true,**	**true,**	**true,**	**true,**	**true,**	**true**
break	= **false,**	**false,**	**false,**	**false,**	**false,**	**false**
miracle	= **true,**	**true,**	**true,**	**true,**	**false,**	**true**
magic	= **false,**	**true,**	**false,**	**true,**	**false,**	**true**
ok	= **true,**	**true,**	**true,**	**true,**	$\acute{v}=\grave{v}$,	**true**
skip	= **false,**	**true,**	**true,**	**true,**	$\acute{v}=\grave{v}$,	**true**
spin	= **true,**	**true,**	**false,**	**false,**	$\acute{v}=\grave{v}$,	**false**

In this formalism, we are able to express not only **chaos** and **havoc**, but **random** and **break** as well, and we can distinguish **ok** from **skip**. An advocate of PF might criticize all weaker formalisms for being unable to make these distinctions. One might well ask if **random** and **break** are useful for any purpose; the answer is that they are just as useful as **havoc**. One might be forgiven for wondering how we are to observe "progress" and "success"; the answer is similar to that for "termination".

Is PF powerful enough? We gave it components that sound almost reasonable; properties like them have all been seriously discussed in the literature. We were tempted to give it a few more components, such as "strangeness" and "charm", to make our point more obvious. If we wish, each of these properties can be observed indirectly via a convention, which is an interpretation of those things that are directly observable. But what is their purpose?

We have kept PS simple. Those who are uncomfortable without an explicit termination indicator are invited to write their programs and other specifications using their favorite termination convention. Then the initial states for which the computation is required to terminate are those for which $(\forall \acute{v} \cdot S \Rightarrow \acute{\sigma})$, i.e. for which the specification implies termination. PS provides the ability to express any convention, but it has none built in.

Conclusion

In all practical circumstances, we can and do observe termination: by means of a termination convention. This is so standard that it may be difficult to realize we are observing a convention, and not termination itself.

We have said that "termination" does not have a physical meaning. We now point out that it does not always have a logical meaning, either. For any semantic formalism and any proof theory, there is a program P such that "P terminates" can neither be proved, nor disproved. There is a model in which it is true, and another in which it is false. For each such program P, the proof theory can be enriched and the model theory constrained to make "P terminates" mean whichever we prefer: true or false. To those who have it firmly fixed in their heads that "termination" means something, this will seem paradoxical.

The simplest of the formalisms, having the properties most convenient for both specifiers and programmers, is PS without a stability convention. Its one "limitation" is its inability to distinguish between **chaos** and **havoc**. But no physical experiment can distinguish between them, either. The distinction is a creation of the more complex formalisms.

Acknowledgement. We thank Cliff Jones and an anonymous referee for helpful suggestions.

References

1. E.W. Dijkstra: A Discipline of Programming. Englewood Cliffs, N.J.: Prentice-Hall 1976
2. E.C.R. Hehner: Predicative Programming. Commun. ACM **27**, 134–151 (1984)
3. E.C.R. Hehner, L.E. Gupta, A.J. Malton: Predicative Methodology. Acta Inf. **23**, 487–505 (1986)
4. C.B. Jones: Systematic Software Development using VDM. London: Prentice-Hall Int. 1986
5. D.L. Parnas: A generalized control structure and its formal definition. C. ACM **26**, 572–581 (1983)
6. W.A. Robison: A Deterministic Trace Semantics for Communicating Sequential Processes. M.A.Sc. Thesis, University of Toronto, 1986

Received October 2, 1986/July 7, 1987