Technical Correspondence

PREDICATIVE PROGRAMMING

Hehner's proposal that predicates can be interchanged with programs and specifications [2] is a good one, but there are older methods that allow us to do that without the "technical difficulties" he notes in his paper. This correspondence discusses those difficulties and advises on the use of other approaches to support predicative programming.

The ability to interchange predicates on the values of state variables with programs or specifications is a consequence of two well-known facts:

(1) The behavior of a program may be described by a set of ordered pairs (x, y) where x and y are possible states of the data, and it is possible for the program to stop in state y after being started in state x. The set of ordered pairs defines a relation, and this approach has been called relational semantics. It has been used by many authors, among them, Majster [3], de Bruijn [1], and Meyer [4]. For the deterministic programs that we usually encounter, the relation is a function. That case has been discussed extensively by Mills [5, 6], Nelson [7], and others. Mills's papers use functional semantics to provide an elegant description of the ideas of structured programming and program verification using the interchangeability of functional specifications and programs.

(2) Any set relevant to this discussion can be characterized by a predicate, which is called its characteristic predicate. That predicate may be used to describe the relational semantics of the program.

Hehner's *new semantics* (NS) can be understood as a variant on *relational semantics* (RS). The notation introduced in [2] is a convention for writing the characteristic predicates. However, the interpretation of the predicates under NS is different from the interpretation of those predicates under RS. Let R be the relation associated with a program by RS, then H, the relation associated with a program by NS, is apparently[1] given by

$$H = R + \{(x, y) \mid \text{termination is not guaranteed when the program is started in state } x\}.$$

We can illustrate the difference in interpretation by several examples. In RS, a program that will not terminate when started in state x is denoted by excluding x from the domain of R; in NS, this is denoted by including the pair (x, y) for all data states y. In RS, a program that never terminates would be characterized by the predicate *false*. In NS, it can only be characterized by *true*. In NS, *true* describes all possible programs, that is, an unrestricted set of programs. In RS, *true* describes the least-restricted program, a program with no restrictions on its behavior. NS offers no predicate that only describes programs that never terminate. The implications of this innovation can be best discussed after reviewing a well-known problem with RS.

When RS is used for deterministic programs, two programs with externally distinguishable behavior will always be represented by distinct relations. For nondeterministic programs this is not the case. Consider two programs X and Y as follows:

- X is guaranteed to terminate for all initial states.
- Y has behavior identical to X except that, for some starting states, it may sometimes, but not always, fail to terminate. When it does terminate, it will only terminate in a state in which X could terminate.

X and Y are described by the same relation (and hence the same predicate) in RS. This is a severe weakness in RS, which has led some workers to conclude that it cannot be used for nondeterministic programs.

Two ways to correct this deficiency have been proposed. Several authors have added a distinguished final state representing nontermination. The characteristic predicate of such relations cannot be a predicate on values of the program variables (in the familiar predicate calculus) because the variables have no values in the special state. One can also supplement the relation with a set, called the competence set in [8], containing the states in which termination is guaranteed. Both approaches are equivalent in the sense that the algebraic properties of the mathematical objects are the same. It is not possible to provide complete descriptions of an unrestricted class of programs by a single predicate on the values of the program variables. Either one uses two predicates, or one adds additional elements that are not program variables.

Although NS associates different predicates with X and Y in most (but not all) cases, it cannot express other distinctions that can be expressed by RS. The specification associated with Y will not reveal that it can only terminate in states where X could terminate. Hehner argues that such distinctions are not useful because nonterminating mechanisms are undesirable. My experience suggests otherwise.

A software engineer should be able to use a program semantics to write specifications that can express the difference between any pair of programs with observably different behavior. Otherwise, one will be forced to overspecify, that is, require something that is not really needed. This can happen with regard to nontermination because there are applications where one would want to either:

(1) Specify that either termination in a restricted set of finite states or nontermination is allowable. We

[1] The relation is not explicitly stated in [1], but the one given here is consistent with all the examples and discussion.

might want to forbid termination in certain states because termination in those states would lead to undesirable effects from subsequent programs.

(2) Specify that termination in certain situations is allowed, but not required. These are cases that will never arise, and it is therefore not appropriate to constrain the behavior.

(3) Specify that termination is required, but that the state is immaterial. We might do this because the program that is executed afterwards will have behavior that is independent of its starting state.

We cannot distinguish these three cases if we use the NS interpretation of the predicates when writing specifications. In fact, we cannot express (1) or (3) at all.

There are other complications. The NS definition of composition is far more complicated than the usual definition of relational composition. A consequence of that definition is that composition in NS is not associative; the properties of the predicates are clearly different from the properties of the programs they are to model. A further complication resulting from the NS interpretation of the predicates comes with the introduction of nondeterministic choice ("or"). Because of the way that NS represents nontermination, it is possible that the "or" of two programs, each of which is required to terminate, will not be required to terminate. Again, the properties of the mathematical structure are not those of the programs it was intended to model.

As Hehner points out, one can avoid some of these difficulties by the introduction of (otherwise) superfluous variables. However, we should ask why one should accept these problems and possible pitfalls when RS supports the same programming methodology without such anomalies. In the sequel we review the methods available for three different classes of programs and specifications.

Deterministic Programs and Specifications

When dealing with deterministic programs, the RS relations are functions, and one can use functional semantics (FS). The weakness of RS, described above, does not arise with FS, and one can use the characteristic predicates of the functions associated with the programs to describe or specify programs. These predicates completely characterize the programs. The definition of composition is simple and well known, and there are none of the technical problems noted above. This suffices for many practical applications because the programs that we usually deal with are deterministic. Mills's work has led to widespread use of FS within the IBM Federal Systems Division.

Nondeterministic Programs and Specifications with "Safe" Domains

There are situations in which we want to deal with nondeterministic programs or use specifications that allow a choice of final states for certain starting states. It is useful to distinguish a restricted class of programs and specifications, in which for all states x, if a program can ever terminate when started in state x, it is certain to terminate when started in state x. We refer to these as having safe domains. In the terminology of [8], programs with safe domains are programs in which the domain of the program is the same as the competence set of the program.

If you restrict your attention to programs or specifications with safe domains, the characteristic predicate of the RS relation provides a complete description of the program or specification. Unfortunately, the set of programs with safe domains is not closed under ";". Programs X and Y may both have safe domains while $X;Y$ does not. (Note that this is a property of the programs and specifications, not a quirk of the formalism used to represent them.) The vast majority of programs and specifications of practical interest are programs with safe domains, and as long as one remains within that class, one can use a single predicate corresponding to the standard relational semantics without fear of mathematical anomalies. Theorems corresponding to those proved in [2] about NS are proved about relational semantics in standard mathematical texts. Because FS is a special case of this method, it is easy to compare deterministic programs with programs in this class and to use FS whenever a safe program is found to be deterministic.

Unrestricted Nondeterministic Programs

An unrestricted class of nondeterministic specifications and programs can be represented by the RS predicate together with an additional predicate giving the set of states in which termination is guaranteed (LDRS). Adding the second predicate is a natural extension to the approach used for programs with safe domains, making it easy to compare safe domain programs with other programs and to determine when a program has a safe domain. The pair of predicates constitutes a representation of an LD-relation. All properties of LD-relations are properties of nondeterministic programs. The necessary theorems about these predicates were proved in [8].

I have made extensive use of the notation introduced in [2], but applied the older interpretation of the predicates as outlined above. It can be used by both experienced programmers and beginners. I have not found any situation in which there is an advantage to using the NS interpretation. Because the properties of the mathematical objects used in FS, RS, and LDRS are exactly those of the programs that they model, one need never worry that one will unintentionally write a program that is not properly represented by the semantics. Such errors can be particularly vexing because the semantics cannot help you in such cases.

Acknowledgments. John McLean and Art Sedgwick made useful suggestions during the preparation of this correspondence.

David L. Parnas
Dept. of Computer Science
University of Victoria
Victoria, B.C. V8W 2Y2 Canada

REFERENCES
1. de Bruijn, N.G. Unpublished reports on the Automath Project, Technische Hogeschool Eindhoven, The Netherlands.
2. Hehner, E.C.R. Predicative programming: Part I. *Commun. ACM 27*, 2 (Feb. 1984), 134–143.
3. Majster-Cederbaum, M. A simple relation between relational and predicate transformer semantics for non deterministic programs. *Inf. Process. Lett. 2*, 4, 5 (Dec. 12, 1980).
4. Meyer, A.R., and Halpern, J.Y. Axiomatic definitions of programming languages: A theoretical assessment. *J. ACM 29*, 2 (Apr. 1982), 555–576.
5. Mills, H.D. The new math of computer programming. *Commun. ACM 18*, 1 (Jan. 1975), 43–48.
6. Mills, H.D. Mathematical foundations of structured programming. *Software Productivity*. Little Brown, Boston, Mass., 1983.
7. Nelson, E. Functional programming analysis. *J. Syst. Softw. 2* (1981), 225–235.
8. Parnas, D.L. A generalized control structure and its formal definition. *Commun. ACM 26*, 8 (Aug. 1983), 572–581.

AUTHOR'S RESPONSE

Parnas's letter shows a number of fundamental misunderstandings. I shall reply on two levels, one concerning the general predicative principle and the other concerning the details of my particular semantics.

A description of X is a true statement about X. "Is Catholic" describes the Pope, and "is Jewish" does not, because the former is true of the Pope and the latter is not. When Parnas states "In RS, such a program would be described by *false*," he is clearly misusing the word "describe," since *false* is not true of anything. Of course, Parnas is free to make any association between programs and predicates that he wishes, but he cannot call his association a descriptive semantics. In my paper, I was careful to follow this general principle: The stronger the predicate, the fewer things it describes, and at the extremes, *true* describes everything and *false* does not describe anything.

That is one half of my "predicative principle." The other half concerns what a program is. In my paper, I do not specify or describe programs. I specify desired computer behavior (activity). In the traditional view, a program is identified with computer behavior, hence it is traditional to talk about the specification of programs. But I am considering programs to be specifications (of desired computer behavior). This view is essential in allowing me to consider programs as predicates and to integrate programming notations with other specification notations. This is a main point of my paper, but Parnas clearly has not taken it to heart. He says "In NS, *true* describes all possible programs," "a program that never terminates," and similar phrases. Of course, all programs terminate; it is the machine activity they specify that may not terminate. Does anyone talk about a nonterminating specification?

A forthcoming paper by C. A. R. Hoare in [1] entitled "Programs Are Predicates" adopts exactly the predicative principle, as in my paper (but with a different choice of programming connectives). For further reading, I recommend it.

Now let me turn to the detail in my semantics that seems to bother Parnas most: the treatment of termination. My specifications (and therefore my programs) speak only of initial and final values of variables and of communication sequences. I have chosen not to speak directly of termination. If we specify that a particular poststate is required, then indirectly we are saying that termination is required. If we specify that an infinite sequence of communications is required, then indirectly we are saying that nontermination is required. (We may even specify that an infinite sequence of communications is required of one process, and that a poststate is required of another. For a comment on that possibility, see Lengauer's letter.) The relation that Parnas associates with a program according to my semantics should have been

$$H = \{(\dot{v}, \vec{v}) \mid \text{for input } \dot{v}, \text{ output } \vec{v} \text{ is acceptable}\}$$

with no mention of termination.

According to Parnas, "Hehner argues that such distinctions (e.g., may terminate versus must not terminate) are not useful because nonterminating mechanisms are undesirable. My experience suggests otherwise." In Part II of my paper, I introduce communication channels and consider nonterminating activity without any difficulty. Parnas must be referring only to Part I (he never mentions communication and references only Part I). The assumption there is that, after an initial input, without communication channels, nothing further is observable unless and until the computation properly terminates. And then, what is observable is not the fact of termination, but the final values of the variables. Without communication channels, nonterminating behavior is unobservable and cannot possibly serve any purpose.

Parnas lists three kinds of specification he might want, and states that my specifications cannot be used for the first and third. The first is to "forbid termination in certain states." This is no problem for my specifications: $\dot{x} \neq 3$ is a specification satisfied only by behavior that does not terminate with $x = 3$. Of course, if we are to observe that a mechanism achieves (implements) this specification, its behavior must terminate in some other state. (We cannot observe that a mechanism's behavior never terminates.) In Parnas's third kind of specification, "termination is required, but . . . the [final] state is immaterial." This time Parnas is right: My specifications can say that the final state is undetermined, but not that termination is also required. However, in the absence of communication channels, such behavior is useless.

The decision not to speak directly about termination was made also by Hoare in his axiomatic semantics and by Dijkstra in his predicate transformer semantics, and with similar consequences. We are unable to distinguish among some or all of the pathological behaviors: abortion, nontermination, undetermined poststate. For example, using Dijkstra's weakest precondition semantics, we cannot say that nontermination is required (not even indirectly). (The reader should note that $\neg wp(S, \mathit{true})$ is interpreted as meaning that no postcondition is

guaranteed, hence termination is not guaranteed, but termination in an arbitrary poststate is a possibility.) These semantics, like mine, were designed for the computations that people want. To study pathology, we may wish to use Parnas's semantics. Or even better, we can have a predicative semantics and speak directly about (non)termination simply by adding a variable for that purpose, exactly as suggested in the appendix to Part I of my paper.

Parnas is wrong to say that "RS supports the same programming methodology." Here is a simple example. The specification (in integer variables n and x)

$$S: \dot{n} \geq 0 \Rightarrow \acute{x} = \dot{n}! \wedge \acute{n} = \dot{n}$$

says that if n is initially nonnegative then the final value of x must be the factorial of n's initial value, and n must be unchanged. According to the predicative semantics of **if-then-else**,

$$(\textbf{if } n > 0 \textbf{ then } (\dot{n} > 0 \Rightarrow \acute{x} = \dot{n}! \wedge \acute{n} = \dot{n})$$
$$\textbf{else } (\dot{n} = 0 \Rightarrow \acute{x} = \dot{n}! \wedge \acute{n} = \dot{n})) \Rightarrow S$$

is a trivial theorem. It says that the problem S can be solved by the use of **if-then-else**, introducing two new problems to be solved. But before we solve them, we have the confidence of a theorem that we have taken a correct step, and a good implementation can tell us immediately if we have made a logic error. This step does not commit us to any particular solutions to the new subproblems, but expresses them in their full generality. Similarly, according to the predicative semantics of assignment and composition (semicolon),

$$(n := n - 1; S; n := n + 1; x := x \times n)$$
$$\Rightarrow (\dot{n} > 0 \Rightarrow \acute{x} = \dot{n}! \wedge \acute{n} = \dot{n})$$
$$(x := 1) \Rightarrow (\dot{n} = 0 \Rightarrow \acute{x} = \dot{n}! \wedge \acute{n} = \dot{n})$$

are theorems that complete the solution. The ability to mix programming and other specification notations this way depends essentially on the predicative principle.

Acknowledgments. Lorene Gupta suggested the example program.

Eric C.R. Hehner
Computer Systems Research Institute
Sanford Fleming Building
University of Toronto
Toronto, Ontario M5S 1A4
Canada

REFERENCES

1. Hoare, C.A.R., and Shepherdson, J.C., Eds. *Mathematical Logic and Programming Languages*. Prentice-Hall, Englewood Cliffs, N.J., 1985.

Hehner provides a semantics that identifies imperative (i.e., Pascal-like) programs with "implementable" predicates that relate in values of program variables v (denoted $\dot{v}$) to out values (denoted $\acute{v}$). Program variables may be ordinary (assignable) variables or channel variables. There is a weakest possible description of the state of a program variable x; let us call it $K(x)$, or "chaos at x." For ordinary variables it is the predicate *true*—nothing may be assumed about the state of the variable; for channel variables it is the assertion that past communications are never modified (see [1] for a formal definition). Hehner employs assertion $K =_{df} \forall x \epsilon v.\ K(x)$, where v is the vector of program variables. His use of chaos deserves some discussion.

Most of Hehner's semantics follows the philosophy of *global chaos*: "If chaos occurs at one variable, it will spread to all others." For instance, if an assignment $x := e$ goes wrong, that is, the in value of expression e is undefined, chaos results at all variables of vector v. Similarly, in the composition $P;Q$, chaos in P makes all of $P;Q$ chaos. For example, for P: $x := 1/0$ and Q: $y := 0$, Hehner's semantics yields $P;Q = Q;P = true$. Composing P and Q spells chaos for both x and y, because of a chaotic assignment of x. At least in the absence of channel variables, $P;Q$ is appropriately implemented by sequential execution of P and then Q. (We shall consider channels later.) Hehner points out that composition is, in some rare cases, not associative.

In the conclusion of Part I, Hehner proposes (but does not adopt) the alternative philosophy of *local chaos*: "If chaos occurs at one program variable, it will spread only to dependent variables." Variable x depends on variable y if an assignment of x uses y. In this philosophy, a chaotic assignment spoils only the target variable. Solely the definition of assignment governs the propagation of chaos. Composition is not concerned with chaos. For our previous programs P and Q, this semantics yields $P;Q = Q;P = (\acute{y} = 0)$. Variable y is spared from chaos, since y does not depend on x. Composition is associative, and its implementation requires a data-flow analysis.

One connective in Hehner's semantics follows the philosophy of local chaos: independent composition $P \parallel Q$. In independent composition, chaos in one program does not spread to the other. Two programs P and Q may only be composed by $\parallel$ if they are independent. P and Q are independent if they manipulate distinct portions v_P and v_Q of vector v. Our programs P: $x := 1/0$ and Q: $y := 0$ can be independently composed to $P \parallel Q = (\acute{y} = 0)$. $P \parallel Q$ may be implemented by parallel execution of P and Q.

In Hehner's semantics, the composition of P and Q does not imply their independent composition. This means that not all parallel executions of P and Q may be replaced by execution in sequence. We would like independent composition to always subsume composition: $P;Q \Rightarrow P \parallel Q$. Hehner's semantics provides the reverse implication: $P \parallel Q \Rightarrow P;Q$ (Theorem 14 in [1]). Best would be the conjunction of both implications: $P \parallel Q = P;Q$. Then, independent composition would be a special case of composition with no purpose other than to simplify the detection of concurrency, as Hehner intended. We accomplish $P \parallel Q = P;Q$ by entirely committing the semantics to one or the other chaos philosophy:

(1) Global chaos—redefine independent composition as follows:

$$P \parallel Q =_{df} (((\neg\forall \dot{v}.\ P{=}K) \wedge (\neg\forall \dot{v}.\ Q{=}K)) \Rightarrow (P(v_P) \wedge Q(v_Q))) \wedge K.$$

Now, independent composition $(P \parallel Q)$ subsumes composition $(P;Q$ and $Q;P)$. To achieve equality of $P \parallel Q$ and $P;Q$, we must also weaken composition

$$P;Q =_{df} (((\neg\forall \dot{v}.\ P{=}K) \wedge (\neg\forall \dot{v}.\ Q{=}K)) \Rightarrow (\exists \dot{v}.\ P^{\dot{v}}_{v} \wedge Q^{\dot{v}}_{v})) \wedge K.$$

This version of composition is again not associative. It reflects a very strict philosophy of global chaos, in which chaos spreads backward from the point of its occurrence to previous program parts. Any arbitrarily interleaved composition of independent P and Q equals their ordinary composition $P;Q$.

(2) Local chaos—keep independent composition as is, and adopt the previously defined local chaos semantics for assignment and ordinary composition. Since channel input $c?x$ and channel output $d!e$ can be expressed as assignments of sequence variables (see [1]), they inherit local chaos semantics from assignment and composition. Essentially, in case of missing input or unsuccessful output, chaos is restricted to the variables involved: c and x, or d, respectively. The semantics of input choice $[a?x \rightarrow P \,\square\, c?y \rightarrow Q]$ requires a similar change. If input is missing on both channels a and c, chaos occurs at a and x, and c and y; the states of all other variables are preserved.

One pleasing property of a semantics where $P \parallel Q = P;Q$, as in both (1) and (2), is that the ordinary composition of independent programs is commutative since it equals their independent composition, which is commutative. This holds, in particular, for independent channel communications. In contrast to [1], $c?x;\ d!1 = d!1;\ c?x$.

Global chaos is the "centralized" chaos philosophy. It yields a less determined, that is, weaker semantics in which a computation may either succeed completely or fail completely. An occurrence of chaos in just one of many independent components may cause abortion of the entire program. A concurrent computation terminates when all its concurrent components have terminated. Concurrency is identified by independent composition and is implemented by traditional techniques; for example, it may be simulated with arbitrarily interleaved sequential execution.

Local chaos is the "distributed" chaos philosophy. It yields a more determined, that is, stronger semantics that takes the notion of partial success to the extreme. Chaos only affects dependent parts of a computation. Only those parts may abort; independent parts must continue. This requires a data-flow analysis. Independent composition becomes obsolete. Its justification was to identify concurrency. Here, concurrency is a by-product of the data-flow analysis. The rest of the semantics inherits one of the properties of independent composition: the potential for unbounded nondeterminism.

Hehner's semantics is partly centralized and partly distributed. Program parts (processes) that are themselves composed by centralized semantics (ordinary composition) may be further composed by distributed semantics (independent composition). Independent composition may not be replaced by ordinary composition. Independent composition identifies concurrency, but its implementation still requires a data-flow analysis: Freedom from chaos of $P \parallel Q$ does not imply freedom from chaos of both P and Q.

We have based our discussion on the assumption that ordinary composition is appropriately implemented by normal (von Neumann) sequential execution. In Hehner's semantics this holds without a doubt in the absence of channel variables. In the presence of channels, a program may execute infinitely without being chaos. Hehner gives the example of a program *ONES*: $(d!1;\ ONES)$ that outputs an infinite sequence of ones on a channel d. Because *ONES* is not chaos, $(ONES;\ x := 2)$ must satisfy $\dot{x} = 2$. Channels suggest the notion of "partial" termination. To support it, we may, as in this example, prefer the implementation to delay certain operations and instead apply an order of execution that is fair to both components of the composition. An implementation with such capabilities gives composition local chaos semantics, and if such an implementation is assumed, composition should be defined accordingly. Then, ordinary composition once again implies independent composition, as we desire.

Christian Lengauer
Dept. of Computer Sciences
University of Texas at Austin
Austin, TX 78712

REFERENCES
1. Hehner, E.C.R. Predicative programming (Parts I and II). *Commun. ACM 27*, 2 (Feb. 1984), 134–151.

AUTHOR'S RESPONSE

I agree with Lengauer. I wish to point out that, in my semantics, $A;B$ can always be implemented as sequential execution (first A, then B) in circumstances where we would expect that implementation (e.g., A is noncommunicating, or A communicates only a finite amount). But, as Lengauer points out, in other circumstances, it is less obvious what the execution should be. When A is an infinite but nonchaotic communicating loop and B does not depend on the variables in A, we may prefer that B be given a fair portion of time. Or, we may prefer sequential execution, so that B is infinitely delayed. I believe my semantics is open to both interpretations.

The main point of my papers is not the specific choice of connectives that I have defined, but the predicative (and prescriptive) approach to the semantics of programs. I am happy to see alternative suggestions, particularly when they have nicer algebraic properties than my definitions.

Eric C.R. Hehner