VAUGHAN R. PRATT

Application of Modal Logic to Programming*

Abstract. The modal logician's notion of possible world and the computer scientist's notion of state of a machine provide a point of commonality which can form the foundation of a logic of action. Extending ordinary modal logic with the calculus of binary relations leads to a very natural logic for describing the behavior of computer programs.

Background

Before dealing with the modal logic connection a few words on the general role of logic in programming may be in order. What makes programming a different activity from operating a calculator is the use of variables in place of concrete values. The sequence of keys pressed by the calculator user describes a *single* computation, while a program describes a *set* of computations, using variables as place-holders for values. The meaning of a variable in a computer program is as for the Tarskian semantics of formulae: the state of the computer memory at the time the program is executed defines an interpretation of the program variables, so that the computer always calculates with concrete values.

In contrast to the computer, the programmer must aspire to a higher order of calculation in convincing himself that his program works as intended on all possible inputs. The programmer's problem is that the variables are not interpreted (do not have particular values) at the time of writing the program. The programmer need not be the only one with this problem; if a program needs to be audited for any reason (e.g. in verifying that a payroll program will not transgress the tax laws) then the auditor inherits the programmer's problem. To deal with the problem one introduces additional calculation rules that work even for incompletely specified values, ranging from simple rules such as evaluating $x-x$ as 0 and $x+0$ as x, up to more elaborate rules such as induction and quantifier elimination. Logic supplies exactly the machinery needed for such symbolic calculation.

Connections between modal logic and programming can be made at two levels, syntactic and semantic. At the syntactic level the modal logician uses such referentially opaque constructs as "necessary" and "possible," or "compulsory" and "permitted", while the computer scientist would like to say that his program always leaves the variable x non-

* This research was supported by NSF Grant No. MSC – 7804338.

-negative, or eventually halts, which can also be viewed as referentially opaque constructs. At the semantic level there exist natural correspondences between what the modal logician calls a possible world and the computer scientist calls a state of a machine, and between the modal logician's relation of accessibility and the computer scientist's program viewed abstractly as a function (or relation in the case of nondeterministic programs) on states.

This paper describes dynamic logic, a system of reasoning about action that represents one possible application of modal logic to programming that makes the above-mentioned connections. This is by no means the only such application, and a brief survey of other applications appears at the end of the paper. The emphasis here on dynamic logic reflects the author's personal involvement with that system.

The origins of dynamic logic are as follows. In the spring of 1974 I was teaching a class on the semantics and axiomatics of programming languages. At the suggestion of one of the students, R. Moore, I considered applying modal logic to a formal treatment of a construct due to C. A. R. Hoare, "$p\{a\}q$", which expresses the notion that if p holds before executing program a then q holds afterwards. Although I was sceptical at first, a weekend with Hughes and Cresswell convinced me that a most harmonious union between modal logic and programs was possible. The union promised to be of interest to computer scientists because of the power and mathematical elegance of the treatment. It also seemed likely to interest modal logicians because it made a well-motivated and potentially very fruitful connection between modal logic and Tarski's calculus of binary relations, a connection which in hindsight should have been studied in detail many years ago. The system was first described in class notes [26] and was later published in 1976 [27]. The epithet "dynamic" was not applied until [12], the term being chosen in preference to some word suggestive of programs in recognition of the potential of the system for reasoning about actions arising in non-programming contexts.

Syntax

A language is a set of expressions. In dynamic logic an expression may be a formula, a term, or a program. Just as propositional calculus contains only formulae, and first-order logic only formulae and terms, propositional dynamic logic (PDL) contains formulae and programs, and first-order dynamic logic contains formulae, terms, and programs.

We shall let the metavariables h, h', ... range over expressions, p, q, r, ... over formulae, x, y, z, ... and f, g, ... over terms (the latter being exclusively for terms not of ground type, i.e. functions and functionals), and a, b, c, ... over programs.

We let $\mathscr{L} = \Phi \cup \Upsilon \cup \Sigma$ range over languages consisting of formulae, terms, and programs respectively.

There is no fixed language associated with dynamic logic. Rather one chooses for study some manageable subset of the following constructions. The research surveyed below deals for the most part with mercifully small such subsets.

Φ: $\top$, $\sim p$, $p \vee q$, $[a]p$, $\langle a \rangle p$, ∞a, $x = y$, ...
Υ: $f(x)$, f_i^x, 0, $x + y$, ...
Σ: $a \cup b$, $a; b$, a^*, a^-, $p?$, $x := ?$, $x := y$, $aa.b$, ...

In addition one may choose to draw on variables of each kind, propositional variables $P, Q, R, \ldots$ ranging over truth values, term variables $X, Y, Z, \ldots$ and $F, G, \ldots$ ranging over individuals and functions, and program variables $A, B, C, \ldots$ ranging over relations on states.

Some of these constructs should be familiar to everyone, particularly $\top$ (*true*) $\sim$ $\vee$ $= 0$ $+$ and $f(x)$ (application of f to x). Some readers will also recognize $\cup$; * $^-$ (the relational calculus constructs of union, composition, ancestral, converse), and $x := y$ (assignment, a programming construct, with x constrained to be a variable). Modal logicians should see through the thin disguises of the dual constructs $[a]p$ and $\langle a \rangle p$ (the role of "a" is to name the particular relation of accessibility intended). In fact only ∞ f_i^x $?$ $:=?$ and $aa.b$ should raise questions in the minds of the majority of the readers. The construct ∞a means that the program a may run forever; f_i^x is the function satisfying $f_i^x(i) = x$ and $f_i^x(j) = f(j)$ for $j \neq i$ (useful for assignments to arrays); $p?$ is a test (useful for synthesizing various conditional constructs in programs); $x := ?$ is *random assignment* (useful for defining quantification); and $aa.b$ is the least a that makes $a = b$ (where b is presumably some program containing free occurrences of the program variable a).

The constructs $x = y$, 0, $x + y$, ... are relevant to dynamic logic only inasmuch as a full fledged system of logic for reasoning about action will need these and many more constructs. We will not mention them further.

We adopt a fairly standard set of syntax conventions, for example reading $\langle a \rangle p \supset q$ as $(\langle a \rangle p) \supset q$ and $p \wedge q \supset r \vee s$ as $(p \wedge q) \supset (r \vee s)$. Spacing will also be used judiciously, so that $p \supset q \supset r \supset s$ is to be read as $(p \supset q) \supset (r \supset s)$.

Semantics

The framework within which we shall embed all our definitions of the meaning of the constructs we consider is based on the one familiar to modal logicians, using possible worlds or *states* as we shall call them. A *semantic structure* for dynamic logic is a quadruple $(\mathscr{L}, W, D, \mu)$ where

$\mathscr{L}$ is a *language*, W a *universe* of states, D a *semantic domain*, and $\mu: \mathscr{L} \to (W \to D)$ a *semantic function* which for each expression $h \in \mathscr{L}$ and state $w \in W$ specifies what h denotes in state w.

When $\mathscr{L}$ consists solely of formulae as in the case of the propositional calculus D need contain only truth values. When $\mathscr{L}$ consists solely of ground terms in integer arithmetic, say, D need contain only integers. When $\mathscr{L}$ consists of ground programs, D need contain only elements (or subsets if nondeterminism is to be treated) of W. When $\mathscr{L}$ consists of combinations of these, D becomes accordingly more complex.

Generalizing a much-used convention, we write $u \models h$ for $\mu(h)(u)$ where h is an expression and u is a state. Thus $u \models p$ will be a truth value as usual, while $u \models x$ will be an element of D, possibly an integer or a function, and $u \models a$ may be a state or a set of states depending on which particular logic we have. (For readers accustomed to thinking in terms of binary relations, the binary relation R implicitly assigned to a by μ in this scheme of things satisfies uRv just when $v \in u \models a$. This assumes the case when $u \models a$ is a subset of W.)

A formula p *has a model* S, or is *satisfiable*, when $u \models p$ is *true* for some $u \in W$ of some semantic structure S. Furthermore p is *valid* when $\sim p$ is not satisfiable. The *theory* of a system with language $\mathscr{L}$ is the set of valid formulae of $\mathscr{L}$.

We are now in a position to identify some familiar and not-so-familiar logics. Propositional calculus consists of propositional variables, $\sim$ and $\vee$ (with other logical connectives being definable in terms of these). D must then contain *true* and *false*, and μ must satisfy the constraints

$$u \models \sim p \equiv u \not\models p$$
$$u \models p \vee q \equiv u \models p \text{ or } u \models q.$$

(Note that "$\sim$" and "$\vee$" are in the *object language*, i.e. the language under study, while "u," "$\models$," "$\equiv$," "or," and the variables p, q, are in the *metalanguage*, the language we are using unquestioningly to communicate with in this paper.)

We may remark that the propositional calculus formula p has a model if and only if it has a model with one state.

The theory of propositional calculus is recursive; more accurately, it is complete (to within log space) in nondeterministic polynomial time. (We include this and other computational complexity results for the sake of those familiar with the terminology, which we shall not define here. See, e.g., [31] or [1].) Furthermore a complete Hilbert-type axiom system can be constructed from an adequate supply of tautologies together with the rule Modus Ponens.

We may now make the transition from propositional calculus to modal logic. From our viewpoint the system **K** of modal logic extends propositional calculus by adding to $\mathscr{L}$ exactly one program variable A and

the construct $\langle a\rangle p$ (where a can only be A), with $u \Vdash a$ being a set of states, with $u \Vdash A$ being otherwise unconstrained, and with μ further satisfying

$$u \Vdash \langle a\rangle p \equiv \exists v \in u \Vdash a \quad \text{such that} \quad v \Vdash p.$$

From the programmer's point of view $u \Vdash a$ is the set of states program a may terminate in when started in state u, while $\langle a\rangle p$ asserts of a state u that if program a is started in u, it may terminate in a state satisfying p. If deterministic programs were to be treated exclusively, $u \Vdash a$ could be taken to be a single state, although then one would need a distinguished "limbo" state to represent nontermination (failure to reach a final state).

We may introduce $[a]p$ as the abbreviation for $\sim\langle a\rangle \sim p$, or (with equivalent effect) as the construct satisfying $u \Vdash [a]p \equiv \forall v \in u \Vdash a,\ v \Vdash p$. As such it is the dual of $\langle a\rangle p$. $[a]p$ asserts of u that if program a is started in u then if and when it halts p will be true.

The satisfiable formula $P \wedge \langle A\rangle \sim P$ demonstrates that a one-state model will not always suffice in satisfying a formula of $\boldsymbol{K}$. This in turn establishes that no formula q of propositional calculus can express $\langle A\rangle P$ in the sense that q has the same truth value as that of $\langle A\rangle P$ in all states of all semantic structures, since such a q would have to be satisfiable in some one-state model, a contradiction.

The theory of $\boldsymbol{K}$ is recursive [18], and in fact complete in polynomial space [20]. A complete axiom system can be obtained by extending a complete axiomatization of propositional calculus with the distributive axiom $[a](p \supset q) \supset [a]p \supset [a]q$ and the rule of Necessitation, from p infer $[a]p$.

Defining the system $\boldsymbol{K}$ via a single variable A leads very naturally to the system $\boldsymbol{K}^*$, which is $\boldsymbol{K}$ with an inexhaustible supply of program variables $A, B, C, \ldots$. $\boldsymbol{K}^*$ by itself is an uninteresting extension of $\boldsymbol{K}$ for about the same reason that propositional calculus without logical connectives is uninteresting. The axiomatization of $\boldsymbol{K}$ serves as an axiomatization of $\boldsymbol{K}^*$ without change. However, $\boldsymbol{K}^*$ leads us to the core of dynamic logic, the integration of program connectives into modal logic. We begin with the three *regular* connectives, $\cup$; and *.

Union. A natural concept for actions is that of having a choice of which action to carry out. The action $a \cup b$ offers the choice of actions a or b. The associated constraint on μ is

$$u \Vdash a \cup b = u \Vdash a \ \cup\ u \Vdash b.$$

The validity $\langle a \cup b\rangle p \equiv \langle a\rangle p \vee \langle b\rangle p$, which we may call the *union axiom*, completely captures union in dynamic logic. This axiom demonstrates that adding union to $\boldsymbol{K}^*$ does not increase expressive power since

every formula of $\boldsymbol{K}^*$ with $\cup$ can be translated to one without $\cup$. On the other hand a certain degree of succinctness is obtained as can be seen by considering $\langle A \cup B\rangle\langle A \cup B\rangle \ldots \langle A \cup B\rangle p$, i.e. $\langle A \cup B\rangle^n p$, which is not expressible by a $\boldsymbol{K}^*$ formula of length less than 2^n, as the reader may verify. (Hint: show that in any such formula q and for any string s of A's and B's of length n there must be a path from the root of q viewed as a tree such that the first n $\langle A\rangle$ and $\langle B\rangle$ connectives encountered along that path correspond to s. Otherwise there would be a model of q consisting of $n+1$ states threaded by a path of A's and B's corresponding to s such that one of those edges could be removed without affecting the value of any of the subformulae of q, so that q would be true when it should not.)

Composition. A familiar concept to programmers is that of executing one program after another; we may execute first a and then b. The *composition* of a and b, written $a; b$, describes the net effect of executing first a and then b. Formally:

$$u \Vdash (a;\ b) = (u \Vdash a) \Vdash b$$

where $U \Vdash b$, for $U \subseteq W$, is the union of the $u \Vdash b$'s for each u in U.

The validity $\langle a;\ b\rangle p \equiv \langle a\rangle\langle b\rangle p$, the *composition axiom*, completely captures composition in dynamic logic. Like the union axiom, the composition axiom demonstrates that ; adds no new expressive power. Unlike the union axiom, which amounts to a distributivity axiom, the composition axiom deals with associativity, and it does not even improve succinctness.

Iteration. In order to get a program to run for a substantial time some way of executing programs repeatedly is called for. The most elementary form of repetition is the *iteration* (or the ancestral, or reflexive transitive closure) of a, which from the programmer's point of view means execution of an action an arbitrary number of times. We write a^*(a-star) for the iteration of a. Formally

$$u \Vdash I = \{u\} \quad (I \text{ is the identity action, needed for the next line})$$
$$u \Vdash a^* = u \Vdash (I \cup a \cup \overbrace{a; a \cup a; a; a \cup \ldots})$$

where the ellipsis is meant to go only as far as the natural numbers (no nonstandard models).

Axioms for iteration are not as easy to come by as for union and composition. In fact when we introduce assignment later we will not be able to get a complete axiomatization of iteration. Without assignment however, we can achieve an axiomatization of iteration as follows.

$$[a^*]p \supset p$$
$$[a^*]p \supset [a]p$$
$$[a^*]p \supset [a^*][a^*]p$$
$$p \wedge [a^*](p \supset [a]p) \supset [a^*]p.$$

The second and third of these may be replaced by $[a^*]p \supset [a][a^*]p$. The fourth may be replaced by the rule, from $p \supset [a]p$ infer $p \supset [a^*]p$.

It is not at all apparent that these axioms generate all the valid formulae of PDL (propositional dynamic logic). The fact that they do was first announced in the Notices of the AMS by K. Segerberg [34]. Later (Jan. 1978) Segerberg found a lacuna in his proof, which he repaired some months after. Meanwhile R. Parikh [24] had worked out what seems to be the first satisfactory proof. The present author [28] and D. Gabbay [11] have both given sketches of completeness proofs.

It is also not at all apparent that every satisfiable formula has a model with finitely many states. This was first shown by M. Fischer and R. Ladner [16], who showed by a filtration argument analogous to the one used to show decidability of monadic predicate calculus that as few as 2^n states sufficed where $n = |p|$. They also showed that at most n formulae and programs needed consideration in determining the existence of a model of that size. This leads to a nondeterministic decision method for satisfiability: guess a structure of the appropriate size and check whether it can be extended to a model of p. The checking can be done in time a polynomial in the size of the model, establishing that the theory of PDL is in NTIME (c^n) (nondeterministic Turing machine exponential time) for some c. What spoils this method for practical computation is that the obvious deterministic version of the algorithm needs to consider up to 2^{2^n} models even when only one program appears in the formula. Fischer and Ladner also showed that there was some $d > 1$ such that no algorithm could test satisfiability in less time than d^n. The author has recently given an algorithm that takes deterministic time c^n for some c [30], meeting the Fischer-Ladner lower bound to within a polynomial.

At this point it might be worth interrupting the development to review the situation. We have introduced two types of expressions, formulae and programs. We have specified a framework within which primitive constructs might be defined, based on the notion of a semantic structure containing a language, a universe of states, a semantic domain, and a meaning function. Using this machinery we have defined as primitives the logical connectives $\sim \vee \langle\rangle \cup$; and *. And we have introduced variables of type formula and program.

Although this gives us most of the material we need for a zero-order logic of programs, it contains essentially nothing mathematically novel. The propositional connectives date back to 1847 (Boole and DeMorgan). The $\langle\rangle$ connective is Lewis's "possibly" connective, generalized in the light of relational semantics on possible worlds to a set of such connectives denoting different relations. And the binary relation connectives $\cup$; and * are also well established in mathematics. Thus no component of what we have developed to date is novel. What *is* novel is the combination, both from the mathematical viewpoint as evidenced by the recent rush of results on the topic, and from the programmer's viewpoint, most

programmers being of the opinion that a considerable degree of novelty is necessary for the development of logics that treat programs that manipulate states.

We now proceed with the development.

Tests. Conditionals in a programming language are usually introduced with "if-then-else." However the rules of reasoning can be simplified by using a "smaller" notion of conditional, the test, which can be used in conjunction with $\cup$ and; to synthesize if-then-else, and with ; and * to form while-do. $x > 0?$ is an instance of a test, as is $j = 0 \vee p(j) = t(k)?$.

A test $p?$ is constructed from a formula p of the logical language. The idea of a test is that a computation may proceed past a test just when that test evaluates to *true* in the current environment, otherwise the computation must block (not reach any final state). Formally:

$$u \models p? = \begin{cases} \{u\} & \text{if } u \models p \\ \{\} & \text{otherwise.} \end{cases}$$

The *test axiom* is $\langle p? \rangle q \equiv p \wedge q$.

Used in conjunction with the regular connectives, tests make it possible to define "if p then a else b" as $(p?;\ a) \cup (\sim p?;\ b)$, and "while p do a" as $(p?;\ a)^*;\ \sim p?$. They also make it possible to eliminate one more logical connective as a primitive construct, by permitting $p \wedge q$ to be the abbreviation of $\langle p? \rangle q$.

Tests introduce no additional complexity to the problem of deciding satisfiability; nor do they compromise completeness of the axiom system; the test axiom is adequate to axiomatize tests. However tests do increase the expressive power of propositional dynamic logic; there is no formula in test-free PDL equivalent to $\langle (p?;\ A)^* \rangle q$, as shown in an interesting argument by Berman and Paterson [5].

Example. The following gives a simple example of the sort of problem PDL is useful for. Consider the two programs "while P do $(A;\ A)$" and "while P do A". (We assume that testing P has no side-effects, that is, does not cause a change of state.) It is the case that if the first program can reach a final state when started in a given state, so can the second. This is true even if A is nondeterministic. (When A is deterministic, "can reach a final state" means "is guaranteed to halt", or "terminates.") For if not, then P must hold after every execution of A, whence it holds after every execution of $A;\ A$.

This valid statement about the relationship between the termination of the respective programs can be easily stated in PDL, as $\langle$*while* P *do* $(A;\ A)\rangle \mathrm{T} \supset \langle$*while* P *do* $A\rangle \mathrm{T}$, or $\sim\langle\langle (P?;\ A;\ A)^*;\ \sim P? \rangle \sim \langle P? \rangle \sim \sim \sim P? \rangle \sim \langle (P?;\ A)^*;\ \sim P? \rangle \sim \langle P? \rangle \sim P$ if we were to expand out *all* our abbreviations (which we obviously wouldn't want to have to do in actual applications).

First order regular dynamic logic

The transition to any first order logic is made when terms are introduced into the language. A term denotes an arbitrary domain element, not merely a truth value as in the case of a formula, or a set of states as in the case of an action.

The one term-related concept that we encounter here in connection with programs is *assignment*. We shall see both random assignment $x:=?$ and specific assignment $x:=y$, where x is a term variable (one of $X, Y, Z, \ldots$) and y is a term. Both depend on the equivalence relation R_x on states; uR_xv holds just when $u \vDash z = v \vDash z$ for all variables z other than x. Their respective definitions are:

$$u \vDash x:=? = \{v | uR_xv\}$$
$$u \vDash x:=y = \{v | uR_xv \text{ and } v \vDash x = u \vDash y\}$$

The main role for $x:=?$ is for defining quantifiers: $\forall x$ is just $[x:=?]$. Specific assignment is a *sine qua non* of conventional programming languages.

The following serve as axioms for these forms of assignment.

R1	$p \supset \forall xp$	when x does not occur free in p
R2	$\forall xp(x) \supset p(y)$	for any term y
S	$[x:=y]p(x) \equiv p(y)$	

These axioms overly simplify matters. Free occurrences of variables are defined in a more complicated way in dynamic logic (for example x occurs free in $[x:=x+1]x = 5$ but not in $[x:=y+1]x = 5$ or $[x:=3; x:=x+1]x = 5$). Once this notion is defined however, $p(y)$ as used in the above context can then be taken as usual to mean $p(x)$ with all free occurrences of x replaced by occurrences of y with the usual precautions.

R1 says that $x:=?$ may not change any variables other than x. R2 says that $x:=?$ must be able to set x to any value namable in the present state by a term y. As such R1 and R2 act as upper and lower bounds respectively on the interpretation of $x:=?$.

Those familiar with complete axiomatizations of first order predicate calculus will have little difficulty proving its axioms as theorems of the system axiomatized as for ***K*** together with R1 and R2. The language should omit program variables and specific assignments, and include the application construct along with term variables of types D_0 (considering $D_0 \subseteq D$ to be the usual notion of the set of individuals in a predicate calculus model) and $D_0^k \to \{true, false\}$. This permits formation of atomic formulae of the form $f(x_1, \ldots, x_k)$, the application of the predicate symbol f (considered here to be a term variable of type $D_0^k \to \{true, false\}$) to k term variables of type D_0. Assignments $p:=?$ must be forbidden. The rule of Generalization common in Hilbert-style systems is here "genera-

lized" to the rule of Necessitation. It follows that our axiom system for this language is complete. This viewpoint of predicate calculus axioms may have some appeal to those comfortable with our use of K to axiomatize change-of state.

If we now include specific assignment, the amount of logic we have introduced at this point more than covers that catered for in [15], one of the classic papers on logics of programs. Not only does it completely subsume everything offered in that paper, but it goes well beyond it in offering constructs to talk about termination and equivalence of programs, as well as permitting more than one such program-oriented construct as subformulae of a single formula. In contrast the language of [15] permitted only program-oriented formulae of the form $p\{a\}q$ (meaning $p \supset [a]q$), a form which could not be a subformula of other formulae.

The question arises as to the adequacy of the axiomatization of assignment in the presence of the regular program connectives. The story on this may be found in [12, 14]. Once one has term variables (including function and predicate symbols), application, logical connectives, $\langle a\rangle p$, * and specific assignment, the theory is complete in Π_1^1 [12]. Hence a finite axiomatization is out of the question. Refining this a little further, even if the language is restricted to formulae of the form $\exists x \exists y [z: = f(z)^*]p$ where x, y, z are term variables, f is a function symbol and p is a formula containing no $\langle\rangle$ construct (and so containing only quantifier-free first-order formulae), the Π_1^1 lower bound still holds for the corresponding theory. If the language is restricted to formulae of the form $[x: = f(x)^*]p$ where p is any first order formula (quantifiers permitted) then the theory is complete in Π_2^0. (This construct is of particular interest from the point of view of [15], which in essence confines itself to such constructs; the ability to prefix "$[a]q$" with "$p \supset$ " where p is another first-order formula leaves the Π_2^0 result unchanged.)

Two approaches to axiomatizing this theory that have been considered are to give enough axioms to permit translation of any problem into a problem in arithmetic augmented with uninterpreted function symbols [14], and to give the infinitary rule, from $p, [a]p, [a][a]p, \ldots$ infer $[a^*]p$, essentially what is done in [23]. Completeness proofs for both approaches have been supplied by their proponents.

The expressive power of this language is no more than that of constructive $\mathscr{L}_{\omega 1\omega}$, as can be seen by expanding $\langle a^*\rangle p$ as $p \vee \langle a\rangle p \vee \langle a\rangle\langle a\rangle p \vee \ldots$ and applying axiom S to eliminate all assignments, and the axiom for tests to eliminate all tests. Meyer and Parikh [22] have shown that when tests are restricted to being first-order formulae, DL is strictly weaker than constructive $\mathscr{L}_{\omega 1\omega}$.

Example. We give an example of a proof of correctness of a program. A traditional example in computer science circles is the problem of com-

puting the factorial function. The program $A := 1; (X > 0?; A := X \times A; X := X - 1)^*; X = 0?$ will initialize an accumulator A to 1, and then while X remains positive multiply the accumulator by X and decrement X. When X becomes 0 the program is permitted to halt.

One claim that can be made for the program is that provided $X \geqslant 0$ initially the program will always halt. We may express this as $X \geqslant 0 \supset \langle a \rangle \top$ where a is the program above. A proof of this might proceed along the following lines.

$N \geqslant 0 \wedge X = N+1 \supset \langle b \rangle X = N$ (where b is the trio of commands within the * part of a)

$N \geqslant 0 \wedge X = N \supset \langle b^* \rangle X = 0$ (appealing to the obvious induction principle)

$X = 0 \supset \langle X = 0? \rangle \top$

$N \geqslant 0 \wedge X = N \supset \langle a \rangle \top$

$X \geqslant 0 \supset \langle a \rangle \top$ (one might argue this by taking N to be X)

In addition to merely establishing termination one might wish to show that whenever the program halts it leaves the factorial of the initial value of X in A, which we can state as $X = N \supset [a] A = N!$ (which is true even if X is initially negative, since then a will never halt. We might prove this as follows.

$A \times X! = N! \supset [X > 0?](A \times X! = N! \wedge X > 0)$

$A \times X! = N! \wedge X > 0 \supset [A := X \times A] A \times (X-1)! = N!$ (using what we know about factorial)

$A \times (X-1)! = N! \supset [X := X - 1] A \times X! = N!$

$A \times X! = N! \supset [b] A \times X! = N!$ (putting the above three pieces together)

$A \times X! = N! \supset [b^*] A \times X! = N!$ (another induction principle)

$X = N \supset [A := 1] A \times X! = N!$ (clearly)

$A \times X! = N! \supset [X = 0?] A = N!$ (taking 0! to be 1)

$X = N \supset [a] A = N!$ (putting the above three pieces together)

A more detailed proof than this would obscure the way a dynamic logic proof proceeds. From a practical point of view this proof already exceeds the level of detail a modern program verifier would demand to be persuaded of the soundness of an argument. A sensible approach in building such a verfier is to have a general purpose algorithm for testing whether each inference in a proof is sound, rather than whether the proof fits the axioms and rules of some axiomatization of the logic. Such an approach permits the user of the verfier to supply shorter proofs — just how short depends on how good the soundness checker is.

The results of the above two proofs imply the result $X = N \wedge N \geqslant 0 \supset \langle a \rangle A = N!$. Although the converse is not strictly speaking true, knowing that the program is deterministic allows us to infer the converse. In

general, if a is deterministic then $\langle a\rangle p \supset [a]p$. Dually, if a is total (has a halting state corresponding to every initial state), then $[a]p \supset \langle a\rangle p$. Assignment, which is both deterministic and total, satisfies $\langle x := y\rangle p \equiv [x := y]p$.

Miscellaneous constructs

Converse. The converse of a, a^-, can be viewed as the program a run backwards. Formally

$$u \models a^- = \{v | u \in v \models a\}.$$

The *converse axioms* are $p \supset [a]\langle a^-\rangle p$ and $p \supset [a^-]\langle a\rangle p$. From these axioms one may prove $[a^{*-}]p \equiv [a^{-*}]p$ and derive the rule, from $p \supset [a]q$ infer $\langle a^-\rangle p \supset q$. Parikh [24] shows that these axioms alone suffice for a complete axiomatization when PDL is augmented with converse.

While converse is not a construct used in ordinary programming, it is of use in reasoning about programs. Programmers sometimes talk about forward and backward reasoning about a program. In forward reasoning one takes an assertion p and a program a that is started in a state satisfying p, and asks what holds when a halts. The strongest such assertion is called the *strongest consequent* of p via a. In backward reasoning one starts with an assertion q and a program a and asks when program a is guaranteed to terminate (if at all) in a state satisfying q. The weakest such condition is called the *weakest antecedent* of q via a.

The weakest antecedent of q via a can readily be seen to be expressed by $[a]q$. What requires a little more thought is that $\langle a^-\rangle p$ expresses the strongest consequent of p via a. The equi-valid formulae $p \supset [a]q$ and $\langle a^-\rangle p \supset q$ illustrate a certain duality between these two concepts.

Loop. The construct ∞a, or $loop_a$, expresses the idea that at leaast one possible computation of a can run for ever ("diverge"). Using the semantics we have seen so far (namely $u \models a$ is a subset of W), defining ∞a can be awkward since the semantics appears to leave no trace of diverging computations. Nevertheless a fair approximation to the notion may be defined thus.

$$\begin{aligned}
&\sim \infty A && (A \text{ atomic})\\
&\infty a \cup b \equiv \infty a \vee \infty b\\
&\infty a;\ b \equiv \infty a \vee \langle a\rangle \infty b\\
&\infty a^* \equiv \langle a^*\rangle \infty a \vee \langle a^\omega\rangle \top
\end{aligned}$$

These are not meant as axioms to be added to PDL (though in fact the second and third could be, although the first goes against the principle that any program should be substitutable for a variable). Rather they define ∞a inductively on the program a. The meaning of $\langle a^\omega\rangle \top$ (not

a PDL construct) is that it is true in state u just when there exists an infinite path of a's starting from u. (Cycles are permitted, so that the number of *distinct* states encountered along such a path need not be infinite.) If models are constrained so that $|u \models a|$ is always finite (the "bounded nondeterminacy" of [8]), Koenig's lemma makes $\langle a^{\omega}\rangle\top$ equivalent to $\forall n\langle a^{n}\rangle\top$, the definition of ∞a^{*} used in [13].

Those familiar with the filtration process of [10] will easily see that ∞A^{*} cannot be expressed in PDL. For suppose p expresses ∞A^{*}. Then arbitrarily long but finite chains of A's supply models of $\sim p$. But the filtration process applied to a sufficiently long such chain (having more than 2^{n} states where n is the length of $\sim p$) will identify two states in the chain, producing a model of ∞A^{*}. Yet filtration for models of PDL formulae preserves model-hood, a contradiction.

Using non-trivial constructions Meyer and Winklmann [21, 37] have shown that the ∞ construct does not add to the expressive power of first-order DL, a quite surprising result.

D. Gabbay [correspondence] has proposed the use of the Grzegorczyk axiom for axiomatizing ∞a^{*}, namely $\sim p \wedge [a^{*}]\big([a^{*}](p \supset [a^{*}]p) \supset p\big) \supset \infty a^{*}$. In [13] D. Harel and the author proposed the strictly stronger axiom $[a^{*}](p \supset \langle a\rangle p) \supset (p \supset \infty a^{*})$ (which actually was given there in the still stronger form $[a^{*}]\big(p \supset (\langle a\rangle p \vee \infty a)\big) \supset (p \supset \infty a^{*})\big)$. It is not known whether this axiom completely axiomatizes ∞.

E. Dijkstra [8] has developed a logic whose central construct is $wp(a, p)$. M. Wand has in effect shown that the weakest model of Dijkstra's axioms assigns to $wp(a, p)$ the definition $[a]p \wedge \langle a\rangle \mathrm{T} \wedge \sim \infty a$, though not using the dynamic logic terminology. In [13] an axiomatization for first-order DL with ∞ is given and shown, to be complete when arithmetically valid formulae may be taken as axioms. Such a system can then be used as a complete axiomatization of the wp construct. These issues are taken up in greater detail in [14].

Array Assignment. An almost universally used construct in programming languages is $f(x) := y$, the assignment of the value of term y to an element of the array f. A mathematically tractable way of viewing this construct is to consider it to be the assignment $f := f_x^y$ where f_x^y is the function derived from f by changing the value of f at x to y.

Recursion. An imperative program may well benefit from being able to call itself recursively. Such a facility may be conveniently defined via the least fixed point construct. If $d(a)$ is a continuous function on relations (continuous in the sense that for all sets X of relations such that X is totally ordered by set inclusion, $\cup d(X) = d(\cup X)$), then d has a least fixed point (least w.r.t. set inclusion), denoted $\alpha a.d(a)$. (Some writers use μ for α here, or write $Y(\lambda a.b)$.) The significance of fixed points

is that the form of a recursive definition of a program b is

$$b = d(b)$$

where $d(b)$ is some program whose meaning depends on b. The program defined by this is clearly meant to be some fixed point of d; the significance of "least" is that the naive way of running such a recursively defined program happens to yield the least fixed point.

While one might expect that adding recursion to the language increases the difficulty of deciding validity, in fact the problem remains within Π_1^1. In fact it would take an inherently intractable construct (e.g. one that worked by appealing to a Π_2^1 oracle) to make the validity problem any harder than it is for * with assignment.

The bulk of the work done on axiomatizing recursion in dynamic logic appears in [14], where an arithmetically complete axiomatization of recursion is given. No satisfactory infinitary rule has been proposed for recursion.

Other applications of modal logic to programming

This article has focused on dynamic logic mainly because it is the system the author is best qualified to describe in detail. There have however been other applications of modal logic to reasoning about programs, and we sketch them briefly here. Van Emde Boas has surveyed most of this work in more detail [9], omitting only the contributions of Burstall, Ashcroft and Schwarz.

The first published connection between modal logic and reasoning about programs appears to have been made in 1974 by R. Burstall [6], who in the closing section of a paper on a method of proving programs correct introduced the constructs "*Sometimes p*" and "*Always p*" in conjunction with the predicate "$At(L)$" where L labelled a point in a program. With these constructs one could say "$\forall n \geq 0\,[Always\,(At(\text{Start})$ implies $N = n)$ implies $Always\,(At(\text{Loop})$ implies $P = 2^{N-n}]$". One would have access not only to axioms dealing with At but also to such familiar axioms as "*Always*$(p$ implies $q)$ and *Sometimes* p implies *Sometimes* q". Burstall pointed out the connection between possible worlds and machine states, and suggested that the system **S5** was the modal-system closest to his program-oriented system.

F. Kroeger [19] has developed a logic of programs based on modal logic in which the atomic commands are treated as propositional variables whose truth in a given world represents the execution of that command when the processor executing the program is in that world. Again possible worlds correspond to processor states.

E. Ashcroft [1] has developed a programming language in which programming constructs are represented with assertions. The purpose is to permit reasoning about programs by manipulating the programs directly rather than indirectly via a separate logic-oriented language. The meaning of the language is specified with the help of first-order modal logic. The semantics uses a Kripke structure with a total ordering on states, so that the extension of each variable can be taken to be the sequence of values the variable takes on as time passes.

P. van Emde Boas and T. Janssen [16, 17] have applied Montague semantics and Montague's "up" and "down" operators (for mapping between extensional and intensional forms of an expression) to define the meaning of "pointer" variables, a notion that arises in some modern programming languages. Constructing a convincing semantics for this notion appears to present obstacles of a magnitude not generally encountered in defining programming constructs, and no alternatives to Montague semantics are known for defining this notion in this generality.

All of the above deals with programs whose initial and final states are deemed the only significant states as far as reasoning about them goes. What the program does in order to get from a given initial state to its final state is considered immaterial. Such logics do not cover the case of programs whose intermediate states are relevant. A program for monitoring a patient's heartbeat, or scheduling other programs for execution, or answering a series of questions, may not be intended to halt, yet its behavior at its intermediate states is very important.

A. Pnueli [25] has applied temporal logic, in particular the constructs Gp (henceforth p) and its dual Fp (eventually p), to the problem of reasoning about such ongoing processes. Although [25] proposes no semantics, Pnueli has indicated (in a talk given in June 1978) that his formulae should take as values sequences of states.

J. Schwarz [33] has applied Burstall's modal notions of *Sometimes* and *Always* to a method of reasoning about systems of processes operating concurrently. Schwarz's semantics differs from other semantics in that it abandons the notion of a single system state, and instead is based on the idea of an *event*, which though not new in computer science is certainly novel in modal logic.

The author has introduced new modalities with non-Kripke semantics [28, 29] to treat such issues while retaining the syntax (but not the semantics) of the binary relation calculus. The semantics is that a program denotes a set of *sequences* of states (rather than a set of pairs of states as per the usual Kripke semantics) while a formula denotes a set of states as usual. This work may be of interest to modal logicians interested in problems clearly within the scope of modal logic yet not amenable to treatment using binary relation semantics.

All of the above work makes explicit the connection with modal logic. There is in addition much more work on logics of programs that could fruitfully make this connection. Two such logics worthy of mention here are Salwicki's algorithmic logic [3, 32] and Constable's logic [7]. Perhaps it should be argued that these logics are already within the domain of modal logic; however the size of the survey has been kept manageably small by inclusion only of those authors making the modal logic connection explicit.

References

[1] E. A. Ashcroft and W. W. Wadge, *Lucid, a nonprocedural language with iteration*, ***Comm. ACM*** 20, 7, July 1977, pp. 519-526.

[2] J. W. de Bakker and W. P. de Roever, *A calculus for recursive program schemes*, in: ***Automata, Languages and Programming*** (ed. Nivat), pp. 167-196, North Holland 1972.

[3] L. A. Banachowski, A. Kreczmar, G. Mirkowska, H. Rasiowa, A. Salwicki, *An introduction to algorithmic logic; Metamathematical Investigations in the Theory of Programs*, in: ***Mathematical Foundations of Computer Science*** (eds. Mazurkiewicz and Pawlak), Banach Center Publications, Warsaw 1977.

[4] S. K. Basu and R. T. Yeh, *Strong verification of programs*, ***IEEE Transaction Software Engineering***, SE-1, 3, pp. 339-345, September 1975.

[5] F. Berman and M. Paterson, *Test-free propositional dynamic logic is strictly weaker than PDL*, ***T. R. 77-10-02, Dept. of Computer Science, Univ. of Washington***, Seattle, November 1977.

[6] R. M. Burstall, *Program proving as hand simulation with a little induction*, ***IFIP Congress***, Stockholm, August 3-10, 1974.

[7] R. L. Constable, *On the theory of programming logics*, ***Proceeding of the 9th Ann. ACM Symposium on Theory of Computing***, 269-285, Boulder, Col., May 1977.

[8] E. W. Dijkstra, ***A Discipline of Programming***, Prentice-Hall 1976.

[9] P. van Emde Boas, *The connection between modal logic and algorithmic logics*, ***Proceedings of the Mathematical Foundations of Computer Science***, Zakopane, Springer-Verlag Lecture Notes in CS 64 (1978), pp. 1-18.

[10] M. J. Fischer and R. E. Ladner, *Propositional modal logic of programs*, ***Proceedings of the 9th Ann. ACM Symposium on Theory of Computing***, Boulder, Col., pp. 286-294, May 1977.

[11] D. Gabbay, *Axiomatizations of logics of programs*, manuscript, November 1977.

[12] D. Harel, A. R. Meyer and V. R. Pratt, *Computability and completeness in logics of programs*, ***Proceedings of the 9th Ann. ACM Symposium on Theory of Computing***, Boulder, Col., pp. 261-268, May 1977.

[13] D. Harel and V. R. Pratt, *Nondeterminism in logics of programs*, ***Proceedings of the 5th Ann. ACM Symposium on Principles of Programming Languages***, Tucson, Arizona, pp. 203-213, January 1978.

[14] D. Harel, *Logics of programs: axiomatics and descriptive power*, Ph. D. thesis, Dept. of EECS, MIT, MIT/LCS/TR-200, May 1978.

[15] C. A. R. Hoare, *An axiomatic basis for computer programming*, ***CACM*** 12 (1969), pp. 576-580.

[16] T. M. V. JANSSEN and P. van EMDE BOAS, *On the proper treatment of referencing, dereferencing and assignment*, ***Proceedings Int. Cong. on Automata, Languages and Programming***, Turku, Springer-Verlag Lectures Notes in CS 52 (1977).

[17] —, *The expressive power of intensional logic in the semantics of programming languages*, ***Proceedings of Mathematical Foundations of Computer Science***, Springer-Verlag Lectures Notes in CS 53(1977).

[18] S. A. KRIPKE, *Semantical analysis of modal logic I: normal modal propositional calculi*, ***Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*** 9 (1963), pp. 67-96.

[19] F. KROEGER, *Logical rules of natural reasoning about programs*, in: ***Automata, Languages and Programming*** 3 (ed. S. Michaelson and R. Milner), pp. 87-98, Edinburgh University Press, 1976.

[20] R. LADNER, *The computational complexity of provability in systems of modal propositional logic*, ***SIAM Journal on Computing*** 6, 3, pp. 467-480, September 1977.

[21] A. R. MEYER, *Equivalence of DL, DL$^+$ and ADL for regular programs with array assignments*, unpublished report, MIT: August 1977.

[22] A. R. MEYER and R. PARIKH, *Definability in dynamic logic*. Talk given at NSF-CBMS Research Conference on the Logic of Computer Programming, Troy, N. Y., May 1978.

[23] G. MIRKOWSKA, *On formalized systems of algorithmic logic*, ***Bulletin de l'Académie Polonaise des Sciences, Série des Sciences Mathématiques, Astronomiques et Physiques*** 22 (1974), pp. 421-428.

[24] R. PARIKH, *A completeness result for PDL*, ***Proceedings Mathematical Foundations of Computer Science***, Zakopane, Springer-Verlag Lectures Notes in CS 64, 1978.

[25] A. PNUELI, *The temporal logic of programs*, ***18th IEEE Symposium on Foundations of Computer Science***, pp. 46-57, October 1977.

[26] V. R. PRATT, *Semantics of programming languages*, Lecture notes for 6.892, April 1974, M. I. T.

[27] —, *Semantical considerations on Floyd-Hoare logic*, ***Proceedings 17th Ann. IEEE Symposium on Foundations of Computer Science***, pp. 109-121, October 1976.

[28] —, *A practical decision method for propositional dynamic logic*, ***Proceedings 10th Ann. ACM Symposium on Theory of Computing***, pp. 326-337, San Diego, California, May 1977.

[29] —, *Process logic*, ***Proceedings 6th Ann. ACM Symposium on Principles of Programming Languages***, January 1979.

[30] —, *A near-optimal method for reasoning about action*, ***MIT LCS Technical Report TM-113***, October 1978.

[31] R. RUSTIN, *Computational complexity*, ***Courant Computer Science Symposium*** 7, Algorithmics Press, New York, N. Y., 1973.

[32] A. SALWICKI, *Formalized algorithmic languages*, ***Bulletin de l'Académie Polonaise des Sciences, Série des Sciences Mathématiques, Astronomiques et Physiques*** 18 (1970).

[33] J. S. SCHWARZ, *Event based reasoning — a system for proving correct termination of programs*, in: ***Automata, Languages and Programming*** 3 (ed. S. Michaelson and R. Milner), pp. 131-146, Edinburgh University Press, 1976.

[34] K. SEGERBERG, *A completeness theorem in the modal logic of programs*, Preliminary report, Notices of the AMS 24, 6, A-552, October 1977.

[35] R. M. SMULLYAN, ***First-Order Logic***, Springer-Verlag, Berlin, 1968.

[36] M. WAND, *A new incompleteness result for Hoare's system*, ***Proceedings 8th ACM Symposium on Theory of Computing***, pp. 87-91, Hershey, Pennsylvania, May 1976.

[37] K. WINKLMANN, *Equivalence of DL and DL$^+$ for regular programs*, manuscript, Lab. for Computer Science, M. I. T. 1978.

LABORATORY FOR COMPUTER SC.
CAMBRIDGE, U. S. A.

Received November 14, 1978.