Review of
Dynamic Logic[*]

Riccardo Pucella

Department of Computer Science
Cornell University

August 7, 2001

Introduction

In the 1960s, as programming languages were being used to write larger programs, those programs became harder to understand, and people began to worry about issues such as correctness, that is, determining whether a program computed what it was supposed to compute. As a consequence, researchers started to look into the pragmatics of programming, leading among others to a criticism of the GOTO statement [2] and the development of structured programming by Knuth and Wirth. These greatly helped writing programs that were easier to understand, but the issue of showing program correct remained. Undoubtedly helped by the fact that programs had a cleaner structure, researchers began investigating formal approaches to proving programs correct, or, in general, proving that programs satisfied properties of interest. Most approaches involved deriving the proof of a property as one was writing the program, taking advantage of the structured way these programs were written [3, 8]. These approaches were formalized, leading to the total or partial correctness assertions of Hoare [10] or the weakest-precondition calculus of Dijkstra [3]. Essentially, the logic of Hoare dealt with assertions $\{A\}P\{B\}$ around a program P, indicating that if A were true, executing program P would result in B being true. Inference rules indicated how to transform assertions about programs into assertions about larger programs. For instance, if $\{A\}P_1\{B\}$ and $\{B\}P_2\{C\}$ were true, then one could infer that $\{A\}P_1;P_2\{B\}$ was true, with the intuitive reading of $P_1;P_2$ as the sequential composition of P_1 and P_2. Many more formal systems along such lines were devised, and they collectively acquired the name *logics of programs*, or *program logics*.

In a 1976 landmark paper, Pratt recognized that many such program logics could best be understood as modal logics, by essentially associating with every program a modal operator [13]. His idea was developed and refined by Fischer and Ladner [6] and others, culminating into a particular form of program logic called *Dynamic Logic*. Basically, the recognition of the relationship between program logic and modal logic allowed researchers to make use of the vast array of results on modal logics.

The book "Dynamic Logic", by Harel, Kozen, and Tiuryn, offers a self-contained introduction to the subject. The earlier treatments on the subject are either dated, such as the survey by Harel [9] giving the state of the field in 1984, or study Dynamic Logic as a non-trivial extension of modal logic [7]. The latter approach is much more abstract and technically involved. In this review, I hope to give a taste for the subject by looking at its simplest incarnation, the propositional variant of Dynamic Logic, called PDL. (The book describes both the propositional and the first-order variants.) Following which, I will return to the structure of the book, and some personal opinions.

[*]D. Harel, D. Kozen, J. Tiuryn, *Dynamic Logic*, MIT Press, 2000, 459pp, ISBN 0262082896.

Propositional Dynamic Logic

Let's start from the basics. Recall (classical) propositional logic: start with a set Φ_0 of primitive propositions $\{p_1, p_2, \ldots\}$, where a primitive proposition can be understood as a basic fact about which we want to reason, such as "it is raining in Ithaca", or "it is sunny in Ithaca". We define the set of formulas of the logic by induction: a primitive proposition is a formula, and if φ and ψ are formula, so are $\neg\varphi$ and $\varphi \wedge \psi$. We define $\varphi \vee \psi$ as an abbreviation for $\neg(\neg\varphi \wedge \neg\psi)$, $\varphi \Rightarrow \psi$ as an abbreviation for $\neg\varphi \vee \psi$, and $\varphi \Leftrightarrow \psi$ as an abbreviation for $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$. To contrast with the logics we will soon introduce, we refer to this logic (and related ones) as *static logics*. Intuitively, such logics are used to reason about "unchanging" conditions. Propositional modal logic is an extension of propositional logic that permits reasoning about modalities. Typical modalities historically studied by philosophers include *necessity*, namely, that a formula φ is necessarily true (written $\Box\varphi$). One can discuss provability in such a framework, interpreting $\Box\varphi$ to mean "φ is provable". Temporal logic is a logic to reason about time, where $\Box\varphi$ is interpreted as "it is always the case that φ" [4]. Epistemic logic is a modal logic to reason about knowledge, where $\Box\varphi$ (often written $K\varphi$) is interpreted as "the agent knows φ" [5]. Modal logic is a general framework for reasoning about such features, and it was Pratt's insight that one could reason about programs in such a setting. The intuition is to associate with every program α a modal operator $[\alpha]$, and to interpret the formula $[\alpha]\varphi$ to mean "All halting executions of program α result in a state satisfying φ". (The program α may have many possible executions if it is nondeterministic.)

As we will be reasoning about programs, it is a good idea to focus on what the programs look like. We start with an extremely simple language in which to express our programs. We take a set $\mathcal{A} = \{a_1, \ldots\}$ of *primitive programs*. These are abstract operations we want our programs to perform. *Regular programs* are formed by sequencing other programs $\alpha_1; \alpha_2$, by taking nondeterministic choices of programs $\alpha_1 \cup \alpha_2$, or by looping α^* (meaning repeating program a 0 or more times, nondeterministically).[1] We define a logic called PDL (for Propositional Dynamic Logic) to reason about such programs. The syntax of the logic distinguishes between programs and formulas, both sets defined by mutual induction. We define the set of programs essentially as above: a primitive program is a program; if α and β are programs, so are $\alpha; \beta$, $\alpha \cup \beta$, and α^*; moreover, if φ is a formula, then $\varphi?$ is a program. We define the set of formulas as follows: a primitive proposition is a formula; if φ and ψ are formulas, so are $\neg\varphi$ and $\varphi \wedge \psi$; if α is a program and φ is a formula, then $[\alpha]\varphi$ is a formula. Intuitively, the meaning of programs and formulas should be clear. We have seen how programs are interpreted. The program $\varphi?$ checks whether formula φ holds at the current state of the program. If it does, the program terminates. Otherwise, it blocks. Formulas keep the interpretation they have in propositional logic, with the added understanding that $[\alpha]\varphi$ means to execute program α and check if φ holds whenever the program halts. We write $\langle\alpha\rangle\varphi$ as an abbreviation for $\neg[\alpha]\neg\varphi$; one reading of $\langle\alpha\rangle\varphi$ is "at least one halting execution of α results in a state satisfying φ."

Note that we can write some fairly involved formulas using this setup. If we do not put any restrictions on the φs that can appear in programs of the form $\varphi?$, we can write formulas such as $[[\alpha]\psi?; \beta]\varphi$, which says that if all halting executions of α result in a state where ψ holds, then all halting executions of β result in a state where φ holds. It seems counterintuitive for our programs to be able to perform speculative execution in that way, especially since such properties have a tendency to be undecidable for any reasonable programming language. If we restrict PDL to only allow as test formulas those in which no modal operator appears (in other words, a formula in a test can only be a boolean combination of primitive propositions), we call the logic *poor test PDL*. In contrast, the unrestricted version is called *rich test PDL*.

At this point, our logic is just a syntax for writing formulas with an intuitive understanding of what they mean. We can formalize our intuitions about the logic by writing down axioms and inference rules—essentially the properties of the logic. Given our intuitive understanding of the meaning of formulas, we can come up with the following axioms for the logic:

1. axioms for propositional logic
2. $[\alpha](\varphi \Rightarrow \psi) \Rightarrow ([\alpha]\varphi \Rightarrow [\alpha]\psi)$
3. $[\alpha](\varphi \wedge \psi) \Leftrightarrow [\alpha]\varphi \wedge [\alpha]\psi$

[1] Hence the name regular programs: consider a program as a regular expression and let L be the language (over $\mathcal{A}$) generated by the regular expression; each sentence in L is a possible trace or execution of the program.

4. $[\alpha \cup \beta]\varphi \Leftrightarrow [\alpha]\varphi \wedge [\beta]\varphi$
5. $[\alpha;\beta]\varphi \Leftrightarrow [\alpha][\beta]\varphi$
6. $[\psi?]\varphi \Leftrightarrow (\psi \Rightarrow \varphi)$
7. $\varphi \wedge [\alpha][\alpha^*]\varphi \Leftrightarrow [\alpha^*]\varphi$
8. $\varphi \wedge [\alpha^*](\varphi \Rightarrow [\alpha]\varphi) \Rightarrow [\alpha^*]\varphi$

The following inference rules are also used:

$$\frac{\varphi \quad \varphi \Rightarrow \psi}{\psi} \qquad\qquad \frac{\varphi}{[\alpha]\varphi}.$$

We say φ is provable (written $\vdash \varphi$) if φ is derivable from the axioms and the inference rules given above.

At this point, note that all we have done is playing with syntax. We have no way of saying whether a given formula is true or not. It was Tarski's great contribution in the 1930s to point out that semantics can be used to discuss truth of formulas in a logic, independently of any axiom system. One gives a semantics for a logic by exhibiting a *model* for the formulas in the logic, telling us how to assign truth values to formulas. Models for PDL are derived from models for general modal logics due to Kripke [12]. (Indeed, this was one reason for the interest in casting program logics in a modal framework.) Essentially, a model is a set of states; think of all the states a program could be in. Programs take us from one state to another, and at every state we have an interpretation function telling us what primitive propositions are true at that state. General formulas will express properties of moving through that state space. Formally, a model M is a tuple (S, π, σ) where S is a set of states, π is an interpretation function assigning a truth value to each primitive proposition p at each state s, i.e. $\pi(s)(p) \in \{\textbf{true}, \textbf{false}\}$, and σ associates to every primitive program a binary relation on the set of states. Intuitively, $(s_1, s_2) \in \sigma(a)$ if executing the primitive program a in state s_1 leads to state s_2. Our first step is to extend σ to all programs by posing:

$$\sigma(\alpha;\beta) = \sigma(\alpha) \circ \sigma(\beta),$$

$$\sigma(\alpha \cup \beta) = \sigma(\alpha) \cup \sigma(\beta),$$

$$\sigma(\alpha^*) = \textstyle\bigcup_{n \geq 0} \sigma(\alpha)^n.$$

For R and S binary relations, we write $R \circ S$ for the relation $\{(u, v) \ : \ \exists w.(u, w) \in R, (w, v) \in S\}$, and R^n is defined inductively with R^0 the identity relation, and $R^{n+1} = R^n \circ R$.

Using these definitions, we define what it means for a formula φ to be true (or *satisfiable*) in state s of a model M, written $(M, s) \models \varphi$, by induction on the structure of φ. Note that my notation is slightly different from the one in the book, but more in keeping with traditional modal logic presentations.

$(M, s) \models p$ for a primitive proposition p if $\pi(s)(p) = \textbf{true}$,

$(M, s) \models \neg\varphi$ if $(M, s) \not\models \varphi$,

$(M, s) \models \varphi \wedge \psi$ if $(M, s) \models \varphi$ and $(M, s) \models \psi$,

$(M, s) \models [\alpha]\varphi$ for a program α if for all s' such that $(s, s') \in \sigma(\alpha)$, $(M, s') \models \varphi$.

A formula $[\alpha]\varphi$ is true at a state s if for all states s' that can be reached by executing the program α, at state s, φ holds. We can verify that $\langle\alpha\rangle\varphi$ holds at a state s if and only if there is at least one state that is reachable by program α from state s such that φ holds in the state, hence justifying our intuitive reading of $\langle\alpha\rangle\varphi$.

If a formula φ is true at all the states of a model M, we say that φ is valid in M and write $M \models \varphi$. If a formula φ is valid in all models, we say φ is valid, and write simply $\models \varphi$.

We now have two distinct ways of reasoning in the logic: syntactically, by using the provability relation, and semantically, by reasoning about the truth of formulas. Ideally, we would like these two approaches to yield equivalent results. The properties relating the $\vdash$ and $\models$ relations are called soundness and completeness. An axiomatization for a logic is *sound* if anything provable is valid: formally, if $\vdash \varphi$ implies $\models \varphi$. This property allows us to safely reason syntactically: anything we can prove will be true. An axiomatization is *complete* if anything valid is in fact provable: formally, if $\models \varphi$ implies $\vdash \varphi$. Completeness guarantees us that if there is anything interesting we want to say, we can in fact derive it syntactically. One of the core foundational results related to PDL is that the axiomatization above is sound and complete for the Kripke models introduced above.

As we noted, the logic we described above treats programs as sequences of abstract primitive programs. One can say much more interesting and precise things by considering actual operations on an actual state. The typical approach is to consider a set of variables and take a state to be an assignment of values to those variables (also known as a *valuation*), where the values are taken from a fixed *domain of computation*. The primitive programs in such a setting are simply assignments of values to variables. Intuitively, the primitive program $x := 3$ will make a transition from any given state to a state which has the same valuation but for the fact that variable x is associated with the value 3. To reason about such programs, we can extend our logic to a first-order logic, simply called Dynamic Logic. On the logic side, we replace the primitive propositions by a first-order vocabulary of predicate and function symbols. A predicate takes the form $r(t_1, \ldots, t_n)$ for terms $t_1, \ldots, t_n$. A term is either a variable or a function symbol applied to other terms. The interpretation of predicate and function symbols is as in first-order predicate logic: they correspond respectively to relations and functions over the domain of computation. We further allow quantification over variables, so that $\forall x.\varphi$ is an allowed formula. (As usual, we write $\exists x.\varphi$ for $\neg\forall x.\neg\varphi$.) For example, if we assume a standard interpretation for the equality predicate, the formula $\forall y.\langle x := 3\rangle y = x$ will be true in any state with a valuation assigning the value 3 to all variables except possibly x. For a full formalization of the logic, its semantics, and its properties, I will at this point refer to the book.

The book

The book is divided in three parts. The first part is meant to make the book essentially self-contained, by providing the necessary background material needed for the presentation of Dynamic Logic. At a full one hundred and fifty pages, it makes up almost half of the book. The second part focuses on the propositional variant of Dynamic Logic, while the third part focuses on the first-order variant.

Chapter 1, **Mathematical Preliminaries**, is the obligatory review of basic mathematical ideas from discrete mathematics, such as sets, relations, graphs, lattices, transfinite ordinals, and set operators.

Chapter 2, **Computability and Complexity**, reviews the relevant topics from the theory of computation. Among others, it looks at computational models including deterministic, nondeterministic, and alternating Turing machines, the characterization of undecidable problems, and reviews the basic complexity classes, discusses the arithmetic and analytic hierarchies, and first-order inductive definability. It also reviews the basic notions of problem reducibility and completeness, and presents a family of tiling problems, shown complete for various complexity classes.

Chapter 3, **Logic**, thoroughly reviews the basic notions of logic. It introduces several classical logical systems: propositional logic, equational logic (the logic of equality), first-order predicate logic, infinitary logic (a variant of predicate logic that allows some infinite expressions), and modal logic. For each system, syntax and semantics are discussed, as well as axiomatizations and elementary results.

Chapter 4, **Reasoning About Programs**, defines the programming framework assumed throughout the book. It defines the kind of programs studied, namely state-based imperative programs, with a focus on the input/output relation of a program. It discusses the program constructs appearing in the study of Dynamic Logic: while programs, regular programs, recursion, r.e. programs, nondeterminism. It defines the notion of partial and total correctness (partial correctness does not stipulate that a program halts, total correctness does), and introduces Hoare Logic.

Chapter 5, **Propositional Dynamic Logic**, starts the study of PDL, the propositional variant of dynamic logic. It gives the syntax and semantics of PDL, discusses the axiomatization given above, proves it is sound with respect to the semantics, and proves various properties of the logic, with a special focus on the iteration operator * that makes the whole logic nontrivial. It also shows how to faithfully encode Hoare Logic in PDL: intuitively, the Hoare Logic assertion $\{\varphi\}\alpha\{\psi\}$ corresponds to the PDL formula $\varphi \Rightarrow [\alpha]\psi$.

Chapter 6, **Filtration and Decidability**, establishes one of the most significant and surprising results for PDL, the so-called *Small Model Theorem*: if a formula φ is satisfiable, then it is satisfiable in a finite model with a small number of states, exponential in the size of φ. A standard technique from modal logic, *filtration*, is used to establish this result—although the technique has to be adapted because of the special nature of the iteration operator. The Small Model Theorem is surprising because PDL is not compact: an infinite set of formulas can be finitely satisfiable while not being satisfiable, because of the iteration operator.[2] The classical example of such a set of formulas is the set $\{\langle\alpha^*\rangle\varphi\} \cup \{\neg\varphi, \neg\langle\alpha\rangle\varphi, \neg\langle\alpha^2\rangle\varphi, \ldots\}$. However, it turns out that the iteration operator, albeit infinitary in nature, is still uniform enough in its effect to give a Small Model Theorem. This theorem yield as a consequence that it is decidable to determine if a formula is satisfiable: just enumerate all the finite models containing up to $2^{|\varphi|}$ states, and check if φ holds in any of them.

Chapter 7, **Deductive Completeness**, establishes that the axiomatization given above is complete with respect to the semantics of PDL. This is achieved by showing that a consistent formula φ is satisfiable. The proof follows the canonical model structure of completeness proofs for modal logic, but with a twist. We first build a canonical *nonstandard* model for φ. A nonstandard model is one where the relation corresponding to the iteration operator, i.e. $\sigma(\alpha^*)$, is not the transitive reflexive closure of $\sigma(\alpha)$; rather, we simply require that $\cup_{n\geq 0}\sigma(\alpha)^n \subseteq \sigma(\alpha^*)$. We then collapse the nonstandard model into a standard model by filtration.

Chapter 8, **Complexity of PDL**, revisits the decidability of the satisfiability problem, showing that the problem has a more efficient algorithm than the naive one presented in Chapter 6, by giving an EXPTIME algorithm for satisfiability. This is basically as good as it gets, as satisfiability for PDL is shown to be EXPTIME-complete.

Chapter 9, **Nonregular PDL**, explores what happens when we allow nonregular operators in programs. An easy result is that any nonregular operator yields a logic strictly more expressive than PDL. Unfortunately, it does not take much for satisfiability to become undecidable. The bulk of the chapter is devoted to establishing a "threshold" between decidable and undecidable extensions. For example, PDL over context-free programs is undecidable.

Chapter 10, **Other Variants of PDL**, examines variants of PDL studied in the literature, including deterministic PDL (where we disallow various forms of nondeterminism, either syntactically or semantically) and automata PDL (where programs are finite-state machines), programs with various restrictions on allowable tests, programs with complementation or intersections, programs with a converse operator, logics with well-foundedness and halting predicates, and extensions to model concurrency.

Chapter 11, **First-Order Dynamic Logic**, starts the study of the first-order variant of Dynamic Logic. As we noted above, first-order Dynamic Logic adds a domain of computation to the logic. States are no longer abstract, but are taken as a valuation for variables. Primitive programs are no longer abstract, but consist of assignment to variables. Variables also appear at the level of formulas, where they can be quantified over as in first-order predicate logic. Also as in first-order predicate logic, we have a first-order vocabulary consisting of predicate and function symbols, interpreted over the domain of computation. The chapter first discusses the classes of programs considered. This requires more care than in PDL because of the added level of details in the syntax of programs. Regular programs are defined as in PDL, but extensions include arrays, stacks, and wildcard assignment (i.e. an assignment of the form $x :=?$). The semantics of first-order Dynamic Logic are given, necessarily more involved than for its propositional variant. Essentially, one defines a first-order structure for the vocabulary, specifying the interpretation of the predicate and function symbols. This structure is turned into a Kripke structure by considering as the states the valuations of the variables. As in PDL, programs are binary relations between states, and formulas are given a meaning at particular state using the interpretation of the predicate and function symbols, as well as the valuation for the variables.

Chapter 12, **Relationships with Static Logics**, investigates an aspect of Dynamic Logic peculiar to its first-order variant. Reasoning can take two forms: uninterpreted (involving properties independent of the domain of computation), and interpreted (where one focuses on a particular domain or class of domains). This dichotomy permeates the rest of the book. Some basic properties of first-order predicate logic are shown to fail to hold: the Löwenhein-Skolem theorem, completeness, and compactness. Comparisons are then made between uninterpreted first-order Dynamic Logic and infinitary logics. To study Dynamic Logic at the interpreted level, the chapter focuses on a particular domain of computation, the natural numbers with the usual arithmetic operations. Comparisons with infinitary logics are also made.

Chapter 13, **Complexity**, addresses the complexity of first-order Dynamic Logic. More precisely, the difficulty of determining the validity of formulas at either the uninterpreted or the interpreted level is studied, for a variety of Dynamic Logics over the programming languages of Chapter 11. The expressiveness results of Chapter 12 can be used

[2] A set of formulas F is satisfiable if there exists a model M and a state s such that $(M, s) \models \varphi$ for every $\varphi \in F$; a set of formulas F if finitely satisfiable if every finite subset of F is satisfiable.

to give rough bounds, by looking at the difficulty of determining validity for various infinitary logics. (Clearly, because Dynamic Logic subsumes first-order predicate logic, validity is undecidable; the question is: how undecidable?) The chapter also introduces the spectral complexity of a programming language. Roughly speaking, this notion provides a measure of the complexity of the halting problem for a programming language. The spectral complexity of the languages introduced in Chapter 11 is investigated.

Chapter 14, **Axiomatization**, studies axiomatizations of first-order Dynamic Logic. By the results of Chapter 13, since the validity problem for both the uninterpreted and interpreted levels of Dynamic Logic are highly undecidable, one cannot hope to find a nice finitary axiomatization. (This is a basic result from mathematical logic.) This does not prevent one to derive an infinitary axiomatization, including, for instance, inference rules with infinitely many premises. Such a complete axiomatization is given for uninterpreted first-order Dynamic Logic. Similar axiomatizations are given for the interpreted level, although in this case completeness is taken to be relative to an arithmetical structure.

Chapter 15, **Expressive Power**, studies the relative expressive power of languages. This can be done for the uninterpreted level, by comparing Dynamic Logics over both languages in terms of logical expressibility. By comparing the expressive power of logics, as opposed to the computational power of programs, one can compare for example deterministic and nondeterministic languages. The fundamental connection between expressive power of logics and spectral complexity is explored. The impact of nondeterminism, bounded or unbounded memory, various kinds of stacks, and wildcard assignments is investigated.

Chapter 16, **Variants of DL**, considers restrictions and extensions of Dynamic Logic. The interest is mainly in questions of expressive on the uninterpreted level. Discussed are Algorithmic Logic (a predecessor of Dynamic Logic), Nonstandard Dynamic Logic (allowing nonstandard models of time by referring only to first-order properties of time when measuring the length of a computation), an extension of Dynamic Logic with well-foundedness and halting assertions, Dynamic Algebra (an abstract algebraic framework corresponding to PDL), probabilistic variants of Dynamic Logic, and extensions to handle concurrency and communication.

Chapter 17, **Other Approaches**, explores topics closely related to Dynamic Logic. Dynamic Logic is related to Temporal Logic; the main differences being that in Temporal Logic, programs are not explicit in the language, but one can reason about intermediate states of a computation. Process Logic is introduced, essentially a combination of both Dynamic Logic and Temporal Logic. Process Logic uses explicit programs like Dynamic Logic, but moreover provides a way to reason about the intermediate states reached by a program through its temporal operators. The μ-calculus is introduced, which uses as its central expressive feature an operator to compute least fixpoints. The propositional variant, known as the modal μ-calculus, subsumes all known variants of PDL, and various forms of Temporal Logics, while having a simpler syntax. The modal μ-calculus, has become popular for the specification and verification of properties of transition systems. Finally, Kleene algebras (the algebra of regular expressions) and its extension with tests, can be used to carry out simple program manipulations. The advantage is that Kleene algebras form a purely equational subsystem, apparently less complex than PDL (given all known complexity theory results).

Overall, the book is well-paced. The one hundred and fifty pages of background material on computability theory and logic may seem daunting, but in the long run will be appreciated. You can skim over many of the background sections if you are familiar with the material, once you have figured out the particular notation and terminology of the authors.

The description of PDL in part II of the book is technical (it is after all, a book on logic), but easy to follow, with every detail put down on paper. Part III on first-order Dynamic Logic requires a notably more careful reading, both because the material is more complex, and because the treatment is denser. It is especially in that part that following the bibliographical references at the end of every chapter becomes useful, even necessary, for a thorough understanding.

The book should be of interest mainly to students and researchers interested in program verification. It offers interesting features for logicians and philosophers as well, as Dynamic Logic is a nontrivial extension of modal logic. As the "programs" studied in Dynamic Logic need not be typical programs, but any formalized notion of action, Dynamic Logic is well suited for reasoning about actions and their effects, with obvious applications to artificial intelligence, and to normative system specification (where it can be taken as a basis for deontic logic).

The book has been used as the basis for graduate courses; depending on the mathematical maturity of the students, a fair amount of material can be covered. The main determining factor being their previous exposure to mathematical logic. As PDL (resp. Dynamic Logic) extends propositional (resp. first-order predicate) logic, both syntactically and semantically, previous exposure greatly helps, as does exposure to modal logic.

As I stated in the introduction, this book has the decisive advantage of focusing exclusively on Dynamic Logic. Other treatments are often studied after a thorough exploration of modal logic, including the temporal variants. This leads to deep, but difficult to follow treatments for the beginner.

Especially interesting is that the book, while being an entry-level introduction to the subject, does point to active areas of research. Particularly relevant are the application of Kleene algebra as a more tractable theory of program transformations; it is currently being used to verify and reason about compiler optimizations [11]. Process Logic is needed if one is interested in reasoning about nonterminating programs, but it has not received much attention lately. Finally, the μ-calculus is currently *en vogue* in the model checking community, where one is interested in verifying transition systems [1], leading to practical considerations driving research. This book can be seen as an introduction to the underlying ideas needed for a thorough understanding of the μ-calculus as a specification language.

References

[1] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[2] E. W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

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

[4] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 995–1072. The MIT Press / Elsevier, 1990.

[5] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. The MIT Press, 1995.

[6] M. J. Fisher and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[7] R. Goldblatt. *Logics of Time and Computation*. CSLI Lecture Notes, No. 7. CSLI, 1992.

[8] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[9] D. Harel. Dynamic logic. In Gabbay and Guenthner, editors, *Handbook of Philosophical Logic. Volume II: Extensions of Classical Logic*, pages 497–604. Reidel, 1984.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 583, 1969.

[11] D. Kozen and M.-C. Patron. Certification of compiler optimizations using Kleene algebra with tests. In *Proc. 1st Int. Conf. Computational Logic (CL2000)*, volume 1861 of *Lecture Notes in Artifical Intelligence*, pages 568–582. Springer-Verlag, 2000.

[12] S. Kripke. A semantical analysis of modal logic I: normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[13] V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings of the 17th Symposium on the Foundations of Computer Science*, pages 109–121. IEEE Computer Society Press, 1976.