15-414: Bug Catching: Automated Program Verification

Lecture Notes on Programs and Contracts

Matt Fredrikson Ruben Martins

Carnegie Mellon University

Lecture 3

1 Introduction

This lecture advances our understanding considerably beyond propositional logic by marching right ahead to understand programs. Our programming language of choice will be an imperative core language with the most important imperative features such as assignments, if-then-else, sequential composition, and while loops. This setting is concrete but simple enough to enable a comprehensive treatment.

This lecture will study *dynamic logic* [Pra76, HKT00] as the logical foundation for programs and program reasoning. Dynamic logic has been used for many programming languages [Koz85, Pel87, DRS+93, BP06, Pla08, ABB+16, Pla17]. In this lecture, we will study it for a simple imperative programming language, which highlights the most important features of reasoning about imperative programs without getting us bogged down in nonessential issues. In passing, this lecture also introduces first-order logic, which is just as important as dynamic logic.

2 Programs

The first thing we do for the sake of concreteness is to fix the programming language as an imperative core while-programming language with assignments, conditional execution, and while loops.

Definition 1 (Program)**.** *Deterministic while programs* are defined by the following grammar (α, β are programs, x is a variable, e is a term, and Q is a formula of arithmetic):

$$\alpha, \beta \ ::= \ x := e \mid ?Q \mid \texttt{if}(Q)\, \alpha\, \texttt{else}\, \beta \mid \alpha; \beta \mid \texttt{while}(Q)\, \alpha$$

Of course, imperative programming languages have other control structures, too, but they are not essential, because they can be defined out of these. For example a repeat-until loop can easily be defined in terms of the while-loop. There is more variation on the data structures that are supported. Here we start very easily just with a single type of, for example, integer-valued variables. As terms e we use addition and multiplication (but subtraction would be fine to add).

Definition 2 (Terms)**.** Terms are defined by the following grammar ($e, \tilde{e}$ are terms, x is a variable, c is a number literal such as 7):

$$e, \tilde{e} \ ::= \ x \mid c \mid e + \tilde{e} \mid e \cdot \tilde{e}$$

Some applications need further arithmetic operators on terms such as subtraction $e - \tilde{e}$, integer division $e \div \tilde{e}$ provided $\tilde{e} \neq 0$, and integer remainder $e \operatorname{mod} \tilde{e}$ provided $\tilde{e} \neq 0$. Subtraction $e - \tilde{e}$ for example is already expressible as $e + (-1) \cdot \tilde{e}$.

The only possibly slightly subtle program construct is the test statement $?Q$, which tests whether the formula Q is true in the current state and aborts program execution discarding the run otherwise. The test statement imposes a condition on the execution of the program and discards runs that do not fit. For example, if program α can only run if the variable x has a nonzero value, then this corresponds to considering the program $?x \neq 0; \alpha$ that first checks that x is nonzero before continuing to run program α.

For example, a program that has absolutely no effect is `skip` which is the same as the trivial test $?\mathit{true}$, because it imposes no conditions on the state of the program. The program `abort` that aborts execution right away is the same as the impossible test $?\mathit{false}$. Out of these primitives, the test statement $?Q$ is also definable as follows:

$$\begin{aligned} \texttt{skip} &\equiv ?\mathit{true} \\ \texttt{abort} &\equiv ?\mathit{false} \\ ?Q &\equiv \texttt{if}(Q)\,\texttt{skip}\,\texttt{else}\,\texttt{abort} \end{aligned}$$

But then we would have to add new statements `skip` and `abort`, which test statements already provide for free.

The test statement is not strictly necessary for our purposes but will come in handy to illustrate concepts of successful termination as in $?\mathit{true}$ versus aborted program runs as in $?\mathit{false}$ versus nonterminating computation $\texttt{while}(\mathit{true})\ \texttt{skip}$.

3 Program Semantics

Now that we have an intuitive sense of what programs in our simple imperative language mean, let's make this more precise. Just as we did for propositional logic in the previous lecture, we will make things precise by defining a semantics that attaches meaning to each syntactic program that we write.

When we gave the semantics for propositional logic, we defined an interpretation I that mapped all atomic propositions to either true or false. Our programs don't mention atomic propositions, but instead the thing that we want to keep track of is the value that variables take as we make assignments and evaluate terms. What domain can variables take values from? For our purposes today and over the next several lectures, we will keep things simple and just assume that all values are integers. A state ω is a function assigning an integer value in $\mathbb{Z}$ to every variable. The set of all states is denoted $\mathcal{S}$.

We now begin to define the meaning of programs, starting with terms. The value that a term e has in a state ω is written $\omega[\![e]\!]$ and defined by simply evaluating when using the concrete (integer) values that the state ω provides for all the variables in term e.

Definition 3 (Semantics of terms)**.** The *semantics of a term* e in a state $\omega \in \mathcal{S}$ is its value $\omega[\![e]\!]$. It is defined inductively by distinguishing the shape of term e as follows:

- $\omega[\![x]\!] = \omega(x)$ for variable x
- $\omega[\![c]\!] = c$ for number literals c
- $\omega[\![e + \tilde{e}]\!] = \omega[\![e]\!] + \omega[\![\tilde{e}]\!]$
- $\omega[\![e \cdot \tilde{e}]\!] = \omega[\![e]\!] \cdot \omega[\![\tilde{e}]\!]$

The semantics of a program α is a relation $[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$ on the set of states $\mathcal{S}$ such that $(\omega, \nu) \in [\![\alpha]\!]$ means that final state ν is reachable from initial state ω by running program α. A relation $[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$ is also the same as a corresponding set of pairs (ω, ν), which explains the notation.

Definition 4 (Transition semantics of programs)**.** Each program α is interpreted semantically as a binary reachability relation $[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$ over states, defined inductively by

1. $[\![x := e]\!] = \{(\omega, \nu) \;:\; \nu = \omega \text{ except that } \nu[\![x]\!] = \omega[\![e]\!]\}$
   The final state ν is identical to the initial state ω except in its interpretation of the variable x, which is changed to the value that e has in initial state ω.

2. $[\![?Q]\!] = \{(\omega, \omega) \;:\; \omega \models Q\}$
   The test $?Q$ stays in its state ω if formula Q holds in ω, otherwise there is no transition.

3. $[\![\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]\!] = \{(\omega, \nu) \;:\; \omega \models Q \text{ and } (\omega, \nu) \in [\![\alpha]\!] \text{ or } \omega \not\models Q \text{ and } (\omega, \nu) \in [\![\beta]\!]\}$
   The $\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$ program runs α if Q is true in the initial state and otherwise runs β.

4. $[\![\alpha; \beta]\!] = [\![\alpha]\!] \circ [\![\beta]\!] = \{(\omega, \nu) \;:\; (\omega, \mu) \in [\![\alpha]\!], (\mu, \nu) \in [\![\beta]\!]\}$
   The relation $[\![\alpha; \beta]\!]$ is the composition $[\![\alpha]\!] \circ [\![\beta]\!]$ of relation $[\![\beta]\!]$ after $[\![\alpha]\!]$ and can, thus, follow any transition of α through any intermediate state μ to a transition of β.

5. $[\![\texttt{while}(Q)\,\alpha]\!] = \big\{(\omega, \nu) \;:\;$ there are an n and states $\mu_0 = \omega, \mu_1, \mu_2, \ldots, \mu_n = \nu$ such that for all $0 \leq i < n$: ① the loop condition is true $\mu_i \models Q$ and ② from state μ_i is state μ_{i+1} reachable by running α so $(\mu_i, \mu_{i+1}) \in [\![\alpha]\!]$ and ③ the loop

condition is false $\mu_n \not\models Q$ in the end$\}$
The $\mathtt{while}(Q)\,\alpha$ loop runs α repeatedly when Q is true and only stops when Q is false. It will not reach any final state in case Q remains true all the time. For example $[\![\mathtt{while}(\mathit{true})\,\alpha]\!] = \emptyset$.

4 Program Contracts

Let's look at a few simple example programs and their precondition/postcondition contracts, which we continue to describe with `@requires` and `@ensures` clauses similar to what we used to do with contracts in the Principles of Imperative Computation course.

```
//@requires(x=a && y=b);
//@ensures (x=b && y=a);
{x:=x+y; y:=x-y; x:=x-y;}
```

What makes this program interesting is that, as the contract clearly expresses, it swaps the values of variables x and y, but it does so without needing any additional memory. That was sometimes important in the old days of limited memory but is also crucial for topics like reversible computation that are a prerequisite to quantum computing. For us, it's just a simple cute example of a program. But what's interesting to observe is that we need two additional logical variables a and b to even just describe the effect of the clever in-place swapping program in a contract. Indeed, this is reminiscent of the fact that a canonical implementation of swapping would first copy the value of x elsewhere, then copy y into x and then the value of the clone back into y.

Requiring x>=y in the following gcd function is just for sake of simplicity.

```
//@requires x>=y && y>0;
//@ensures x mod a = 0 && y mod a = 0;
//@ensures \forall s (s>0 && x mod s = 0 && y mod s = 0 -> a mod s = 0);
{
    a := x;
    b := y;
    while (b!=0)
    {
        t := a mod b;
        a := b;
        b := t;
    }
}
```

What is not just for the sake of simplicity is the need in the second postcondition to not just say that the resulting value a divides the two inputs x and a but that it also is the greatest such divisor. So for all other divisors s, a is greater or equal s or, in fact, even the other divisor s divides the greatest common divisor resulting from a in the end.

5 The Logical Meaning of a Contract

Apparently, in order to have any chance at all at making sense of the above program contracts, we need to understand more than propositional logic. For one thing, our impoverished view of the world as just atomic propositions p, q, r that are true or false depending on some arbitrary interpretation I is insufficient.

For the gcd program, for example, it is not sufficient to consider $x \geq y$ as an atomic proposition p and $y > 0$ as an atomic proposition q and $x > 0$ as an atomic proposition r. If we did that, an interpretation I could happily interpret $I(p) = \mathit{true}$ and $I(q) = \mathit{true}$ but $I(r) = \mathit{false}$, which is impossible for the concrete arithmetic. For the swap program, it is not sufficient to consider $x = a$ as an atomic proposition p and $y = b$ as an atomic proposition q and $x = y$ as an atomic proposition r and $a = b$ as an atomic proposition s. For if that were the case, then nothing would prevent us from considering an interpretation I in which $I(p) = \mathit{true}$ and $I(q) = \mathit{true}$ and $I(r) = \mathit{true}$ but $I(s) = \mathit{false}$, because it is quite obviously impossible for $x = a, y = b, x = y$ to be true without $a = b$ being true in the same interpretation as well.

Consequently, it will become important for program contracts to consider logical formulas in which concrete terms like $x, x \cdot y, x + a$ or $x \bmod a$ and so on occur and mean exactly what they do in integer arithmetic. Likewise, by the formula $x = y$ we exactly mean the equality comparison of terms x and y and by $x \geq y$ we exactly mean that the value of x is greater-or-equal to the value of y. These are *interpreted* because we have a fixed interpretation in mind for equality ($=$) and greater-or-equal comparison ($\geq$) and addition ($+$) and multiplication ($\cdot$) and so on.

The precise rendition of a contract for the greatest common divisor also inevitably needs a universal quantifier to say that, among all other divisors, the gcd is the greatest. Consequently, we will also find it crucial to extend propositional logic to first-order logic, which comes with universal quantifiers $\forall x\, P$ to say that P is true for all values of variable x. It also supports existential quantifiers $\exists x\, P$ to say that P is true for at least one value of variable x.

So okay, this first-order logic with arithmetic seems much more useful than propositional logic to make sense of contracts. But do they provide us with all we need to understand a program contract?

Given one particular value for each of the variables, first-order logic formulas are either true or false (much like, in an interpretation I, propositional logic formulas are either true or false). But what makes programs most interesting is that the truth of such a first-order logic formula used in, say, a postcondition will depend on the current state of the program. The postcondition may even change its truth-value. It might be false in the initial state but will become true in the final state of the program. In fact, that's often how it works in programs/ The gcd program does not start with the correct answer in the result variable `a` but merely ends up with the correct gcd answer in `a` at the end of the loop.

First-order logic is not very good at that. Just like propositional logic, first-order logic is a static logic, so its formulas will be either true or false in an state/interpretation. But they do not provide any ways of referring to what has been true before a program ran or what will be true after the program did. It is this dynamics, so behavior of change, that calls for *dynamic logic*.

Dynamic logic crucially provides modalities that talk about what is true after a program runs. The modal formula $[\alpha]P$ expresses that the formula P is true after all runs of program α. That formula $[\alpha]P$ is true in a state if it is indeed the case that all states reached after running program α satisfy the postcondition P. We can use it to rigorously express what contracts mean. But let's first officially introduce the language of dynamic logic.

6 Dynamic Logic

Definition 5 (DL formula)**.** The *formulas of dynamic logic* (DL) are defined by the grammar (where P, Q are DL formulas, $e, \tilde{e}$ terms, x is a variable, α a program):

$$P, Q \ ::= \ e = \tilde{e} \mid e \geq \tilde{e} \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid P \leftrightarrow Q \mid \forall x\, P \mid \exists x\, P \mid [\alpha]P \mid \langle\alpha\rangle P$$

The propositional connectives such as $\wedge$ for "and" as well as $\vee$ for "or" mean what they already mean in propositional logic. The equality $e = \tilde{e}$ and greater-or-equal comparison $e \geq \tilde{e}$ also exactly mean that the terms e and $\tilde{e}$ on both sides are evaluated and compared for equality or greater-or-equalness, respectively. This is what distinguishes DL from propositional logic already, because $e = \tilde{e}$ and $e \geq \tilde{e}$ are atomic formulas that do not have arbitrary truth-values that are up to an interpretation I to determine as in propositional logic. Instead, they exactly mean equality and greater-or-equal comparison.

The universal quantifier in $\forall x\, P$ and the existential quantifier in $\exists x\, P$ quantify over all (in the case of $\forall$), or over some (in the case of $\exists$) value of the variable x. But it will be quite important to settle on the domain of values that both quantifiers range over. In most of our applications, this will be the set of integers $\mathbb{Z}$, but other domains are of interest, too.

Most importantly, and indeed the defining characteristic of dynamic logic, are the box modality in $[\alpha]P$ and the diamond modality in $\langle\alpha\rangle P$. The modal formula $[\alpha]P$ is true in a state iff the final states of all runs of program α beginning in that final state satisfy the postcondition P. Likewise the modal formula $\langle\alpha\rangle P$ is true in a state iff there is a final state for at least one run of program α beginning in that final state that satisfies the postcondition P. So $[\alpha]P$ expresses that P is true after all runs of α whereas $\langle\alpha\rangle P$ expresses that P is true after at least one run of α.

7 Contracts in Dynamic Logic

Since the box modality in $[\alpha]P$ expresses that formula P holds after all runs of program α, we can use it directly to express the `@ensures` postconditions of the gcd program. Let `gcd` be the gcd program from above and `postdiv` as well as `postgrt` its two conditions from the two @ensures clauses:

$$\begin{aligned}\mathtt{gcd} &\equiv a := x; b := y; \mathtt{while}(p \neq 0)\,\{t := a \operatorname{mod} b; a := b; b := t\}\\ \mathtt{postdiv} &\equiv x \operatorname{mod} a = 0 \land y \operatorname{mod} a = 0\\ \mathtt{postgrt} &\equiv \forall s\,(s > 0 \land x \operatorname{mod} s = 0 \land y \operatorname{mod} s = 0 \rightarrow a \operatorname{mod} s = 0)\end{aligned}$$

With these abbreviations and the box modalities of dynamic logic it suddenly is a piece of cake to express that the first @ensures postcondition holds after all program runs:

$$[\mathtt{gcd}]\mathtt{postdiv}$$

It is also really easy to express the second @ensures postcondition:

$$[\mathtt{gcd}]\mathtt{postgrt}$$

Well, maybe it would have been better if we had expressed both @ensures clauses at once. How do we do that again?

Well, if we want to say that both postconditions are true after running gcd and the logic is closed under all operators including conjunction, we can simply use the conjunction of both formulas for the job:

$$[\mathtt{gcd}]\mathtt{postdiv} \land [\mathtt{gcd}]\mathtt{postgrt}$$

This formula means that postdiv is true after all runs of gcd and that postgrt is also true after all runs of gcd. Maybe it would have been better to simultaneously state both postconditions at once? Well, that would have been the formula

$$[\mathtt{gcd}](\mathtt{postdiv} \land \mathtt{postgrt})$$

which says that the conjunction of postdiv and postgrt is true after all runs of gcd. Which formula is better now?

Well that depends. For one thing, both are perfectly equivalent, because that is what it means for a formula to be true after all runs of a program. That means the following biimplication in dynamic logic is valid so true in all states:

$$[\mathtt{gcd}]\mathtt{postdiv} \land [\mathtt{gcd}]\mathtt{postgrt} \;\leftrightarrow\; [\mathtt{gcd}](\mathtt{postdiv} \land \mathtt{postgrt})$$

Now that we have worried so much about how to state the postcondition in a lot of different equivalent ways, the question is whether the following formula or any of its equivalent forms is actually always true?

$$[\mathtt{gcd}](\mathtt{postdiv} \land \mathtt{postgrt})$$

Well, of course not, because we forgot to take the program's precondition from the @requires clause into account, which the program assumes to hold in the initial state.

But that is really easy in logic because we can simply use implication for the job of expressing such an assumption:

$$x \geq y \land y > 0 \rightarrow [\texttt{gcd}](\texttt{postdiv} \land \texttt{postgrt})$$

And, indeed, this formula will now turn out to be valid, so true in all states. In particular, in every initial state it is true that if that initial state satisfies the `@requires` preconditions $x \geq y \land y > 0$, then all states reached after running the `gcd` program will satisfy the `@ensures` postconditions $\texttt{postdiv} \land \texttt{postgrt}$. If the initial state does not satisfy the precondition, then the implication does not claim anything, because it makes an assumption about the initial state that apparently is not presently met.

Expressing the contract for the swapping program as a formula in dynamic logic yields:

$$x = a \land y = b \rightarrow [x := x + y; y := x - y; x := x - y](x = b \land y = a)$$

Notice how these dynamic logic formulas make it absolutely precise what the meaning of a program contract is. Well, at least after we define the semantics of dynamic logic formulas, which is our next challenge.

8 Dynamic Logic Semantics

Unlike in propositional logic where everything has a static meaning once and for all, imperative programs are known for changing state and changing the values of variables. Thus, the value of a variable depends on the state, and the state may change as the program is running. For example the assignment $x := x+1$ will move from an initial state ω to a new state ν that has a different value for the variable x, namely exactly such that $\nu(x) = \omega + 1$ while no other variables change their value. But the point is that as imperative programs change state, the meaning of variables in dynamic logic will depend on the state. When x used to have, say, the value 5 in state ω then it will, instead, have the value 6 in the state ν reached from initial ω by running program $x := x + 1$.

We will use the same set of states to define the semantics of DL formulas that we did for programs earlier. In fact, we really must use the same states because DL formulas contain programs in the box and diamond operators, so we need to reason about how programs change states at the beginning of execution into new ones when they terminate. Recall that the semantics of a program α was defined as a relation $[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$ on the set of states $\mathcal{S}$ such that $(\omega, \nu) \in [\![\alpha]\!]$ means that final state ν is reachable from initial state ω by running program α. Likewise, the semantics of terms $\omega[\![e]\!]$ is defined by evaluating the term using the concrete values for variables given in ω.

The semantics of dynamic logic is like that of propositional logic for propositional connectives $\land, \lor, \neg, \rightarrow, \leftrightarrow$ and like that of another influential logic, first-order logic, for quantifiers $\forall$ and $\exists$, extended with a semantics for the modalities $[\alpha]$ and $\langle\alpha\rangle$.

Definition 6 (Semantics of dynamic logic)**.** The DL formula P is true in state ω, written $\omega \models P$, as inductively defined by distinguishing the shape of formula P:

1. $\omega \models e = \tilde{e}$ iff $\omega[\![e]\!] = \omega[\![\tilde{e}]\!]$
2. $\omega \models e \geq \tilde{e}$ iff $\omega[\![e]\!] \geq \omega[\![\tilde{e}]\!]$
3. $\omega \models P \land Q$ iff $\omega \models P$ and $\omega \models Q$.
4. $\omega \models P \lor Q$ iff $\omega \models P$ or $\omega \models Q$.
5. $\omega \models \neg P$ iff $\omega \not\models P$, i.e. it is not the case that $\omega \models P$.
6. $\omega \models P \to Q$ iff $\omega \not\models P$ or $\omega \models Q$.
7. $\omega \models P \leftrightarrow Q$ iff both are true or both false, i.e., it is either the case that both $\omega \models P$ and $\omega \models Q$ or it is the case that $\omega \not\models P$ and $\omega \not\models Q$.
8. $\omega \models \forall x\, P$ iff $\nu \models P$ for all states ν that only differ from ω in the value of variable x.
9. $\omega \models \exists x\, P$ iff $\nu \models P$ for at least one state ν that only differs from ω in the value of variable x.
10. $\omega \models [\alpha]P$ iff $\nu \models P$ for all (final) states ν reachable by running program α from initial state ω, i.e. $(\omega, \nu) \in [\![\alpha]\!]$.
11. $\omega \models \langle\alpha\rangle P$ iff there is at least one (final) state ν reachable by running program α from initial state ω, i.e. $(\omega, \nu) \in [\![\alpha]\!]$ for which $\nu \models P$ holds.

Lemma 7 (Duality). *Dynamic logic satisfies the duality principle that for all programs α and all formulas P the following formula is valid:*

$$[\alpha]P \leftrightarrow \neg\langle\alpha\rangle\neg P$$

This validity is quite similar to the fact that that the following formula is valid

$$\forall x\, P \leftrightarrow \neg\exists x\, \neg P$$

Lemma 8 (Determinism). *The programs α from Def. 1 are* deterministic, *that is, for every initial state ω there is at most one final state ν such that $(\omega, \nu) \in [\![\alpha]\!]$.*

Proof. The proof is by induction on the structure of the program α and a good exercise. □

Because of determinacy, dynamic logic for the deterministic programs from Def. 1 also satisfy another particularly close relationship of the box and the diamond modality:

Lemma 9 (Deterministic program modality relation). *Because the programs α from Def. 1 are* deterministic, *they make the following formula valid for all formulas P:*

$$\langle\alpha\rangle P \to [\alpha]P$$

Proof. Because of Lemma 8, deterministic while programs from Def. 1 are deterministic as their name already suggests. Consequently, there is at most one final state. That is why $\langle\alpha\rangle P \rightarrow [\alpha]P$ is valid for deterministic programs, because if P holds in one final state then it already holds in all final states, because there is at most one final state by Lemma 8. □

Colloquially, we also refer to this lemma as the "one for all" principle. We will occasionally have reason to work with a more general notion of programs that is no longer deterministic, so we should carefully mark all uses of this determinism principle to avoid getting confused about which results depend on determinism.

References

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Matthias Ulbrich, editors. *Deductive Software Verification – The KeY Book*, volume 10001 of *LNCS*. Springer, 2016.

[BP06] Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *LNCS*, pages 266–280. Springer, 2006. doi:10.1007/11814771_23.

[DRS+93] Rainer Drexler, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, Werner Stephan, and Andreas Wolpers. The KIV system: A tool for formal program development. In *STACS*, pages 704–705, 1993.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[Koz85] Dexter Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985.

[Pel87] David Peleg. Concurrent dynamic logic. *J. ACM*, 34(2):450–479, 1987.

[Pla08] André Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, 2008. doi:10.1007/s10817-008-9103-8.

[Pla17] André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, Switzerland, 2017. URL: http://www.springer.com/978-3-319-63587-3.

[Pra76] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. pages 109–121, 1976.