Formal Methods in Software Engineering: Computer-aided Verification

Course Notes for CISC422/853

Winter 2008/09

©Juergen Dingel

School of Computing

Queen's University

Kingston, Ontario

February 2009

Contents

List of Figures

Chapter 1

CTL model checking

CTL model checking uses a temporal logic called computation tree logic (CTL) as specification logic.

1.1 CTL

1.1.1 Syntax

CTL formulas are defined by the following BNF

$$\begin{array}{rcl} \varphi & ::= & \mathit{ff} \mid \mathit{tt} \mid p \mid (\neg\,\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid \\ & & \mathbf{AX}\,\varphi \mid \mathbf{EX}\,\varphi \mid \mathbf{AG}\,\varphi \mid \mathbf{EG}\,\varphi \mid \mathbf{AF}\,\varphi \mid \mathbf{EF}\,\varphi \mid \\ & & \mathbf{A}[\varphi_1\ \mathbf{U}\ \varphi_2] \mid \mathbf{E}[\varphi_1\ \mathbf{U}\ \varphi_2] \end{array}$$

where *ff* and *ff* denote "false" and "true" respectively and p is an atomic proposition, that is, an undevisible boolean expression that can be evaluated in any state. We assume that all atomic propositions are collected into a set AP. Note that every temporal connective is a pair of letters. The first one ('**A**' or '**E**') can be thought of as quantification over the set of paths from the current state. The second one ('**X**','**G**','**F**', or '**U**') can be thought of as quantification over the states in a selected path. Each pair of letters thus represents a nested quantification, the first one over paths, the second over states. The following table indicates the intuitive meaning of each pair.

$\mathbf{AX}\ \varphi$	"Along all paths, in the next state, φ holds"
$\mathbf{EX}\ \varphi$	"Along at least one path, in the next state, φ holds"
$\mathbf{AG}\ \varphi$	"Along all paths, in all future states, φ holds" "Along all paths, φ holds globally"
$\mathbf{EG}\ \varphi$	"Along at least one path, in all future states, φ holds" "Along at least one path, φ holds globally"
$\mathbf{AF}\ \varphi$	"Along all paths, in some future state, φ holds", or "Along all paths, φ holds eventually"
$\mathbf{EF}\ \varphi$	"Along at least one path, in some future state, φ holds", or "Along at least one path, φ holds eventually"
$\mathbf{A}\Big[\varphi_1\ \mathbf{U}\ \varphi_2\Big]$	"Along all paths, φ_1 holds at least until φ_2 does"
$\mathbf{E}\Big[\varphi_1\ \mathbf{U}\ \varphi_2\Big]$	"Along at least one path, φ_1 holds at least until φ_2 does"

The binding priorities of the new connectives generalize the ones for propositional logic.

$$\begin{array}{cl} \neg\ , \mathbf{AX}, \mathbf{EX}, \mathbf{AG}, \mathbf{EG}, \mathbf{AF}, \mathbf{EF} & \text{bind most tightly} \\ \wedge, \vee & \\ \rightarrow & \\ \leftrightarrow, \mathbf{AU}, \mathbf{EU} & \text{bind least tightly} \end{array}$$

1.1.2 Semantics

Formulas are interpreted over interpreted finite state machines. Given an iFSA $M = (S, S_0, L, \delta, F)$, a state s, and a CTL formula φ, the satisfaction relation $(M, s) \models \varphi$ is defined to be the smallest relation that satisfies:

$$\begin{array}{lcl}
(M, s) & \models & tt \\
(M, s) & \models & p \text{ if } eval(p, s) = true \\
(M, s) & \models & \neg\, \varphi_1 \text{ if not } (M, s) \models \varphi_1 \\
(M, s) & \models & \varphi_1 \wedge \varphi_2 \text{ if } (M, s) \models \varphi_1 \text{ and } (M, s) \models \varphi_2 \\
(M, s) & \models & \varphi_1 \vee \varphi_2 \text{ if } (M, s) \models \varphi_1 \text{ or } (M, s) \models \varphi_2 \\
(M, s) & \models & \varphi_1 \rightarrow \varphi_2 \text{ if not } (M, s) \models \varphi_1 \text{ or } (M, s) \models \varphi_2 \\
(M, s) & \models & \mathbf{AX}\ \varphi \text{ if for all } s' \text{ such that } (s, l, s') \in \delta \text{ for some } l \in L, \\
 & & \text{we have } (M, s') \models \varphi \\
(M, s) & \models & \mathbf{EX}\ \varphi \text{ if for some } s' \text{ such that } (s, l, s') \in \delta \text{ for some } l \in L, \\
 & & \text{we have} (M, s') \models \varphi \\
(M, s) & \models & \mathbf{AG}\ \varphi \text{ if for all runs } s_1 s_2 s_3 \ldots \text{ in } M \text{ such that } s = s_1 \text{ we have} \\
 & & (M, s_i) \models \varphi \text{ for all } i \geq 1 \\
(M, s) & \models & \mathbf{EG}\ \varphi \text{ if for some runs } s_1 s_2 s_3 \ldots \text{ in } M \text{ such that } s = s_1 \text{ we have} \\
 & & (M, s_i) \models \varphi \text{ for all } i \geq 1 \\
(M, s) & \models & \mathbf{AF}\ \varphi \text{ if for all runs } s_1 s_2 s_3 \ldots \text{ in } M \text{ such that } s = s_1 \\
 & & \text{there exists } i \geq 1 \text{ such that } (M, s_i) \models \varphi
\end{array}$$

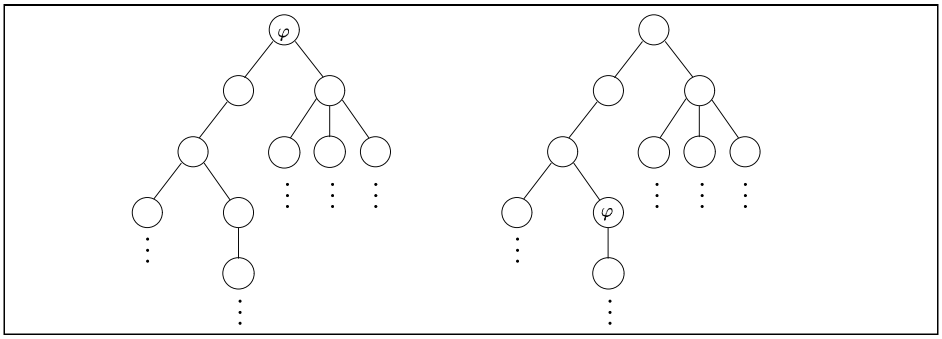
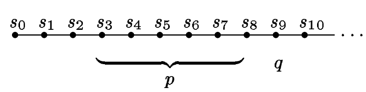


Figure 1.1: Beginnings of two systems whose initial states satisfy **EF** φ.

$$
\begin{array}{lll}
(M, s) & \models & \mathbf{EF}\ \varphi \text{ if for some run } s_1 s_2 s_3 \ldots \text{ in } M \text{ such that } s = s_1 \\
& & \text{there exists } i \geq 1 \text{ such that } (M, s_i) \models \varphi \\
(M, s) & \models & \mathbf{A}\big[\varphi_1\ \mathbf{U}\ \varphi_2\big] \text{ if for all runs } s_1 s_2 s_3 \ldots \text{ in } M \text{ such that } s = s_1 \\
& & \text{there exists some } i \geq 1 \text{ such that } (M, s_i) \models \varphi_2, \text{ and} \\
& & \text{for all } 1 \leq j < i, \text{ we have } (M, s_j) \models \varphi_1 \\
(M, s) & \models & \mathbf{E}\big[\varphi_1\ \mathbf{U}\ \varphi_2\big] \text{ if for some run } s_1 s_2 s_3 \ldots \text{ in } M \text{ such that } s = s_1 \\
& & \text{there exists some } i \geq 1 \text{ such that } (M, s_i) \models \varphi_2, \text{ and} \\
& & \text{for all } 1 \leq j < i, \text{ we have } (M, s_j) \models \varphi_1
\end{array}
$$

The clauses involving propositional connectives only offer no surprises. To illustrate the temporal connectives, it is useful to unwind the state machine into a so-called *computation tree*. The advantage of this representation is that the computation paths of a system can be directly read off.

The computation trees in Figures 1.1, 1.2, 1.3, and 1.4, illustrate the four unary temporal connectives. More precisely, for each connective we give one or two examples of a system in form of a computation tree whose initial state satisfies the formula built using that connective. To illustrate the until operator, consider the following computation path.



Each of the states s_3 to s_9 satisfies $\big[p\ \mathbf{U}\ q\big]$, while the states s_0 to s_2 do not.

Safety and liveness properties: a fundamental distinction

Software properties in general and CTL formulas in particular can be partitioned into two categories:

- Safety properties: Informally, a safety property is a property that states that "nothing bad ever happens" [Lam77]. For instance, deadlock freedom

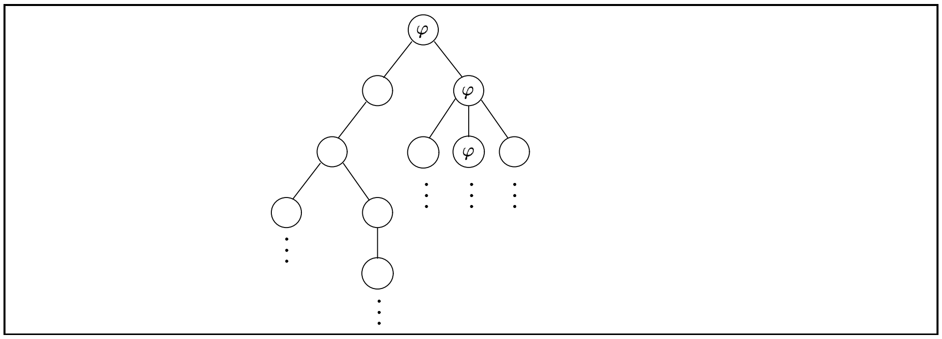


Figure 1.2: Beginning of a system whose initial state satisfies **EG** φ.

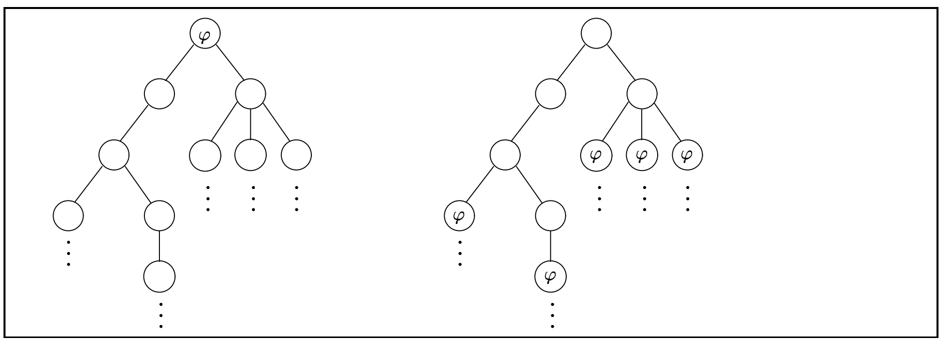


Figure 1.3: Beginnings of two systems whose initial states satisfy **AF** φ.

is a safety property. In CTL, safety properties are thus related to the temporal connectives **AG** and **EG**. Examples of safety properties in CTL include: **AG** $x \neq 0$ and **EG** $(doorOpen \rightarrow fanOff)$.

A safety property is violated iff the system has a (finite or infinite) execution $s_0, s_1, s_2 \ldots$ with state s_i such that "something bad has happened in s_i". Then, the sequence of states $s_0, s_1, \ldots, s_i$ is a counter example. Safety properties, therefore, always have finite counter examples.

- Liveness properties: In contrast, a liveness property expresses that "something good will eventually happen". As before, "eventually" here means after an unknown, arbitrary but finite number of steps. For instance, termination is a liveness property. In CTL, liveness properties are related to the temporal connectives **AF** and **EF**. Examples of liveness properties in CTL include: **AF** $x \neq 0$ and **AG** $(request \rightarrow$ **AF** $granted)$.

  A liveness property is violated iff the system has an infinite execution $s_0, s_1, s_2 \ldots$ along which "the good thing never happens". In this case, the entire execution constitutes the counter example. In other words, liveness

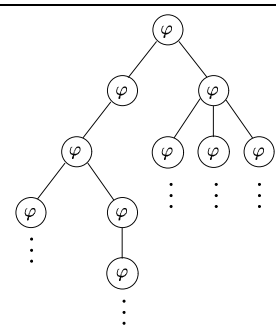


Figure 1.4: Beginning of a system whose initial state satisfies **AG** φ.

properties always have infinite counter examples.

It turns out that this classification is "exhaustive" in the sense that every property can be expressed through the combination of a safety and a liveness property [AS85]. In other words, when reasoning about the correctness of a system, safety and liveness properties are the only kinds of properties you need to worry about.

Schemas of useful CTL formulas

We list a few examples for the kinds of formulas that are often used.

- "It is possible to become super user":

$$\mathbf{EF}\ \mathit{superUser}$$

- "It is always possible to become super user":

$$\mathbf{AG}\ \mathbf{EF}\ \mathit{superUser}$$

- "A request for some resource will always eventually be acknowledged":

$$\mathbf{AG}\ (\mathit{requested} \rightarrow \mathbf{AF}\ \mathit{acknowledged})$$

- "Along every computation path, enabled always holds eventually":

$$\mathbf{AG}\ \mathbf{AF}\ \mathit{enabled}$$

  Note that this means that *enabled* will hold infinitely often along every computation path.

- "A system will never deadlock":

$$\mathbf{AG}\ \neg\ \mathit{deadlocked}$$

- "From every hypertext page it is always possible to get to a hypertext page named 'Home' ":

$$\mathbf{AG}\ \mathbf{EF}\ \mathit{name}{=}\text{'}\mathit{Home}\text{'}$$

- "When picking up the phone it is possible to never receive a dial tone":

$$\mathbf{AG}\ \big(\mathit{rcvPickedUp} \rightarrow \mathbf{EG} \neg\ \mathit{dialTone}\big)$$

- "An upwards traveling elevator at the second floor will not change its direction when it has passengers wishing to go the fifth floor":

$$\mathbf{AG}\ \Big(\mathit{floor}{=}2 \wedge \mathit{direction}{=}\mathit{up} \wedge \mathit{Button5Pressed} \rightarrow \mathbf{A}\big[\mathit{direction}{=}\mathit{up}\ \mathbf{U}\ \mathit{floor}{=}5\big]\Big)$$

Equivalences

We already know that conjunction and disjunction are dual to each other, that is, $(\varphi \vee \psi) \leftrightarrow \neg\,(\neg\,\varphi \wedge \neg\,\psi)$. Similarly, universal quantification and existential quantification in predicate logic are dual, that is, $(\exists x.\varphi) \leftrightarrow \neg\,(\forall x.\neg\,\varphi)$. It turns out that the temporal operators **AG** and **EF** are also dual to each other.

$$\begin{aligned} \neg\,\mathbf{AG}\ \varphi &\leftrightarrow \mathbf{EF}\ \neg\,\varphi \\ \neg\,\mathbf{EF}\ \varphi &\leftrightarrow \mathbf{AG}\ \neg\,\varphi \end{aligned}$$

The next state operator X is its own dual.

$$\neg\,\mathbf{AX}\ \varphi \leftrightarrow \mathbf{EX}\ \neg\,\varphi$$

A path π contains at least one state in which φ holds if and only if tt holds along π until φ does. Consequently, the eventuality operator **F** can be expressed in terms of the until operator **U**.

$$\begin{aligned} \mathbf{AF}\ \varphi &\leftrightarrow \mathbf{A}\big[tt\ \mathbf{U}\ \varphi\big] \\ \mathbf{EF}\ \varphi &\leftrightarrow \mathbf{E}\big[tt\ \mathbf{U}\ \varphi\big] \end{aligned}$$

Finally, the operator **AU** can be expressed in terms of negation, disjunction, **EU**, and **EG**.

$$\mathbf{A}\big[\varphi\ \mathbf{U}\ \psi\big] \quad\leftrightarrow\quad \neg\,\big(\mathbf{E}\big[\neg\,\varphi_2\ \mathbf{U}\ (\neg\,\varphi_1 \wedge \neg\,\varphi_2)\big] \vee \mathbf{EG}\neg\,\varphi_2\big)$$

These equations indicate that there is redundancy between the temporal connectives. We can restrict our attention to an adequate set of connectives to remove this kind of redundancy and to identify minimal sets of connectives from which all other connectives can be obtained.

Definition 1.1.1 *Given a logic L, we say that a set C of connectives of L is* adequate *for L, if all connectives in L can be expressed in terms of connectives in C.*

Example 1.1.1 *1. The set of connectives* $\{\neg\,, \wedge\}$ *is adequate for propositional logic. To see this, consider the following equivalences:*

- $(P \vee Q) \leftrightarrow (\neg\,(\neg\, P \wedge \neg\, Q))$
- $(P \rightarrow Q) \leftrightarrow (\neg\, P \vee Q)$
- $(P \leftrightarrow Q) \leftrightarrow ((P \rightarrow Q) \wedge (Q \rightarrow P))$

2. The set of connectives $\{\neg\,, \vee, \forall\}$ *is adequate for predicate logic.*

Exercise 1.1.1

1. *Show that the set* $\{\mathbf{AU}, \mathbf{EU}, \mathbf{EX}\}$ *is adequate for all temporal connectives in CTL, that is, express the remaining temporal connectives in terms of negation and* **AU**, **EU**, *and* **EX**.

2. *Express* $\mathbf{A}[p\ \mathbf{U}\ q]$ *in terms of* $\neg\, p$, $\neg\, q$, $\wedge$, $\vee$, $\neg$, **EU**, *and* **EG**.

There are lots of adequate sets of connectives for CTL. The following theorem singles out an adequate set that we will use in Section 1.3 when constructing the model checking algorithm.

Theorem 1 *The set of operators false,* $\neg$, $\wedge$, **EX**, **AF**, *and* **EU** *are adequate for all connectives in CTL.*

We conclude this section with a final list of equations.

$$\begin{aligned}
\mathbf{AG}\ \varphi &\leftrightarrow \varphi \wedge \mathbf{AX}\ \mathbf{AG}\ \varphi \\
\mathbf{EG}\ \varphi &\leftrightarrow \varphi \wedge \mathbf{EX}\ \mathbf{EG}\ \varphi \qquad (1.1)\\
\mathbf{AF}\ \varphi &\leftrightarrow \varphi \vee \mathbf{AX}\ \mathbf{AF}\ \varphi \qquad (1.2)\\
\mathbf{EF}\ \varphi &\leftrightarrow \varphi \vee \mathbf{EX}\ \mathbf{EF}\ \varphi \\
\mathbf{A}[\varphi\ \mathbf{U}\ \psi] &\leftrightarrow \psi \vee (\varphi \wedge \mathbf{AX}\ \mathbf{A}[\varphi\ \mathbf{U}\ \psi]) \\
\mathbf{E}[\varphi\ \mathbf{U}\ \psi] &\leftrightarrow \psi \vee (\varphi \wedge \mathbf{EX}\ \mathbf{E}[\varphi\ \mathbf{U}\ \psi]) \qquad (1.3)
\end{aligned}$$

The equation for $\mathbf{AG}\ \varphi$ captures the fact that $\mathbf{AG}\ \varphi$ holds in the current state s if and only if

- φ holds in s, and
- $\mathbf{AX}\ \mathbf{AG}\ \varphi$ holds in s, that is, for all possible next states s' $\mathbf{AG}\ \varphi$ holds in s'.

The equation for $\mathbf{AF}\ \varphi$, on the other hand, expresses that $\mathbf{AF}\ \varphi$ holds in the current state s if and only if

- either φ holds in s, or

- $\mathbf{AX}\ \mathbf{AF}\ \varphi$ holds in s, that is, for all possible next states s', $\mathbf{AF}\ \varphi$ holds in s'.

The computation trees in Figure 1.3 illustrate both cases. Finally, the equation for $\mathbf{A}[\varphi\ \mathbf{U}\ \psi]$ expresses that that $\mathbf{A}[\varphi\ \mathbf{U}\ \psi]$ holds in the current state s if and only if

- ψ holds in s, or
- φ holds in s and $\mathbf{AX}\ \mathbf{A}[\varphi\ \mathbf{U}\ \psi]$ holds in s, that is, for all possible next states s', $\mathbf{A}[\varphi\ \mathbf{U}\ \psi]$ holds in s'.

Similar arguments apply to the equations for $\mathbf{EF}\ \varphi$ $\mathbf{EG}\ \varphi$, and $\mathbf{E}[\varphi\ \mathbf{U}\ \psi]$ respectively.

Notice how each equation describes each connective in terms of itself. Intuitively, for each equation, the formula on the right is obtained by "unwinding" the formula on the left once — a process not unlike, for instance, unwinding a **while** loop into the sequential composition of a conditional and the same loop:

$$\mathbf{while}\ b\ \mathbf{do}\ C \quad = \quad \mathbf{if}\ b\ \mathbf{then}\ C\ ;\mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{end}$$

Indeed, just like in denotational semantics, where the behaviour of a **while** loop is described in terms of all its unwindings, the meaning of each of the temporal connectives can be described in terms of all its unwindings. This description is called *fixed point semantics*. In Section 1.3 we will see how it forms the mathematical basis of our model checking algorithm.

1.2 Example: Mutual exclusion

At this point, we have discussed state machines and CTL and thus have both inputs to a CTL model checker in place. Before we give more detail on how precisely the model checker works, let us look at an example showing how all the notions introduced so far fit together.

Suppose the processes C_i in the concurrent program

$$C \quad = \quad \mathbf{cobegin}\ C_0 \| \ldots \| C_{n-1}\ \mathbf{end}$$

all share a resource, such as a printer, a database or a file on a disk. To ensure consistency of the resource, it may be necessary to prevent multiple processes from updating the resource simultaneously. To solve this problem, we identify so-called *critical sections* and *non-critical sections* in the code of each process and restrict access to the resource using shared variables and synchronization statements such that at most one process is executing its critical region at any given time. This property is called *mutual exclusion*. We will assume that each location in each process is labeled and that each process C_i has the following shape

$$C_i \quad = \quad l_i{:}\mathbf{while}\ \mathit{true}\ \mathbf{do}$$

$$\begin{aligned} nc_i\colon\ & C_{i,nc} \\ cr_i\colon\ & C_{i,cr} \\ \textbf{end}\colon & l_i' \end{aligned}$$

where the labels nc_i and cr_i indicate the non-critical and the critical sections of C_i respectively. Note that we have to assume that all critical sections $C_{i,cr}$ always terminate. The non-critical sections $C_{i,nc}$, however, may or may not terminate. Moreover, since the state machine corresponding to C must be finite for model checking to be applicable, we assume that all critical and non-critical sections have a finite state space.

We want to modify each process C_i such that access to the critical sections is mutually exclusive. The main idea is to place the critical section of each process between an *entry* and an *exit protocol*. The entry protocol in process C_i will protect the critical section $C_{i,cr}$ by checking if other processes are currently executing their critical section. The exit protocol will notify the other processes of the termination of the critical section and possibly allow other processes to enter their critical section.

1.2.1 First attempt

Consider the program below. A variable *turn* that ranges over the numbers from 0 to $n-1$ is used to indicate which process will be allowed to enter its critical region. Let $\oplus$ denote addition modulo n.

$$\begin{aligned} C_{i,1} \quad = \quad & \textbf{while}\ \mathit{true}\ \textbf{do} \\ & \quad nc_i\colon\ C_{i,nc}; \\ & \quad en_i\colon\ \textbf{await}\ \mathit{turn} = i; \\ & \quad cr_i\colon\ C_{i,cr}; \\ & \quad ex_i\colon\ \mathit{turn} := \mathit{turn} \oplus 1 \\ & \textbf{end} \end{aligned}$$

The above algorithm is also called *round robin* algorithm. Before we use a CTL model checker to verify that the modified system satisfies mutual exclusion, we need to express the mutual exclusion in CTL.

- If we're dealing with only two processes, the formula

  $$\mathit{mutex} \quad = \quad \mathbf{AG} \neg (pc_0 = cr_0 \wedge pc_1 = cr_1)$$

  expresses that they cannot be in their critical section at the same time. This formula generalizes to more than two processes in the obvious way.

After this modification the execution of the critical sections is indeed mutually exclusive. More precisely, if M_C is the finite state machine corresponding to C for some fixed n, and s is an arbitrary initial state of M_C, then

$$(M_C, s) \models \mathit{mutex}$$

holds.

Unfortunately, this solution suffers a drawback. If the execution of the non-critical section $C_{i,nc}$ of process i never terminates, process $i \oplus 1$ will never get permission to enter its critical section. In other words, a protocol must satisfy more properties than mutual exclusion to be considered a good solution. Thus, before we proceed, let us collect the other properties that we want the protocol to satisfy. It turns out that there are two more properties besides mutual exclusion.

- **Eventual entry** Whenever any process wants to enter its critical section, it will eventually be permitted to do so.

$$\mathit{evtlEntry} \quad = \quad \mathbf{AG}\ (pc_i = en_i \rightarrow \mathbf{AF}\ pc_i = cr_i)$$

- **Deadlock freedom** The system never deadlocks, more precisely, it is never the case that all processes get stuck forever in their entry protocols.

$$\mathit{noDeadlock} \quad = \quad \mathbf{AG}\ \neg\ (\mathit{blocked}_0 \wedge \ldots \wedge \mathit{blocked}_{n-1})$$

where $\mathit{blocked}_i = \mathbf{AG}(pc_i = en_i)$. Note that this definition of deadlock is slightly different from "all processes are stuck because they are waiting for each other".

Exercise 1.2.1 *What is the logical relationship (if any) between eventual entry and deadlock freedom?*

1.2.2 Second attempt

The problem with our first attempt above is that a process can be given the right to enter its critical section without being interested in entering it. To remedy this, we introduce a boolean variable req_i which, when set, indicates that process i is interested in entering its critical section.

$$\begin{array}{rcl} C_{i,2} & = & \textbf{while}\ \mathit{true}\ \textbf{do} \\ & & \quad nc_i:\ C_{i,nc}; \\ & & \quad en_{i,1}:\ req_i := tt; \\ & & \quad en_{i,2}:\ \textbf{await}\ turn = i; \\ & & \quad cr_i:\ C_{i,cr}; \\ & & \quad ex_{i,1}:\ req_i := ff; \\ & & \quad ex_{i,2}:\ turn := f(i) \\ & & \textbf{end} \end{array}$$

where

$$f(i) \quad = \quad \begin{cases} j, & j \text{ is smallest interested process, ie,} \\ & j \text{ is smallest } k \text{ for which } req_k = tt. \\ i, & \text{if there is no interested process.} \end{cases}$$

Unfortunately, this approach doesn't completely correct the problem encountered in the previous attempt. Consider for instance the 2-process system

$$C \quad = \quad turn := 0\ ;\ \textbf{cobegin}\ req_0 := ff\ ;\ C_{0,2}\ \|\ req_1 := ff\ ;\ C_{1,2}\ \textbf{end}$$

where $C_{0,2}$ has a non-terminating non-cricital section and $C_{1,2}$ has an empty non-critical section, that is, $C_{(0,2),nc} = C_{(1,2),nc} =$ **skip**. This system has an execution that ends in the following trace

pc_0	req_0	pc_1	req_1	$turn$
nc_0	ff	nc_1	ff	0
nc_0	ff	$en_{1,1}$	ff	0
nc_0	ff	$en_{1,2}$	tt	0
nc_0	ff	$en_{1,2}$	tt	0

$\vdots$

Variable never gets set to point the interested process, because process $C_{0,2}$ never leaves its non-critical section. In other words, this system still does not satisfy eventual entry, that is,

$$(M_C, s) \not\models \mathit{evtlEntry}$$

where M_C models C above.

1.2.3 Third attempt

We abandon the idea of the single shared variable *turn* granting access. Instead, we start out very naively and allow process i to enter its critical region if there is no other interested process. For simplicity, we assume for the moment that we are dealing with 2 processes only.

$$\begin{array}{rcl} C_{i,3} & = & \textbf{while } \mathit{true} \textbf{ do} \\ & & \quad nc_i\text{: } C_{i,nc}\textbf{;} \\ & & \quad en_{i,1}\text{: } req_i := tt\textbf{;} \\ & & \quad en_{i,2}\text{: } \textbf{await } \neg\, req_{i\oplus 1}\textbf{;} \\ & & \quad cr_i\text{: } C_{i,cr}\textbf{;} \\ & & \quad ex_i\text{: } req_i := ff \\ & & \textbf{end} \end{array}$$

This ensures mutual exclusion, but now the system can deadlock, that is,

$$(M_C, s) \not\models \mathit{noDeadlock}.$$

To see this, consider the execution below

pc_0	req_0	pc_1	req_1
nc_0	ff	nc_1	ff
nc_0	ff	$en_{1,1}$	ff
nc_0	ff	$en_{1,2}$	tt
$en_{0,1}$	ff	$en_{1,2}$	tt
$en_{0,2}$	tt	$en_{1,2}$	tt

$\vdots$

in which both processes move out of their non-critical sections immediately and express interest in entering their critical section at roughly the same time.

1.2.4 Fourth attempt

The problem with the above solution is that both process can execute their entry protocol at the same time and then get blocked at the **await** statement. To remedy this, we introduce a new variable *last*, that, intuitively, whenever both processes are blocked, "breaks the tie" between them and allows one of them to proceed. The resulting solution is called the *tie-breaker algorithm*, or also *Peterson's algorithm*. Let $x_1, x_2 := e_1, e_2$ denote a multiple assignment statement expressing that x_1 and x_2 are updated with the values of e_1 and e_2 respectively at exactly the same time in one atomic step.

$$
\begin{array}{lcl}
C_{i,4} & = & \textbf{while } \mathit{true} \textbf{ do} \\
& & \quad nc_i\colon\ C_{i,nc}; \\
& & \quad en_{i,1}\colon\ \mathit{req}_i, \mathit{last} := \mathit{tt}, i; \\
& & \quad en_{i,2}\colon\ \textbf{await } \left(\neg\, \mathit{req}_{i\oplus 1} \vee \mathit{last} \neq i\right); \\
& & \quad cr_i\colon\ C_{i,cr}; \\
& & \quad ex_i\colon\ \mathit{req}_i := \mathit{ff} \\
& & \textbf{end}
\end{array}
$$

Note that variable *last* is shared. This attempt finally works. Mutual exclusion, eventual entry and deadlock freedom are all satisfied. Moreover, it also scales to arbitrary numbers of processes.

However, being the perfectionists that we are, we are still dissatisfied. The multiple assignment statement used in $C_{i,4}$ is hard, if not impossible, to implement on realistic machines. We will try to replace it with two simple assignments executed sequentially. This leaves us with two versions depending on which of the two assignments is executed first. Perhaps surprisingly, it turns out that these two versions are not identical.

$$
\begin{array}{lcl}
C'_{i,4} & = & \textbf{while } \mathit{true} \textbf{ do} \\
& & \quad nc_i\colon\ C_{i,nc}; \\
& & \quad en_{i,1}\colon\ \mathit{req}_i := \mathit{tt}; \\
& & \quad en_{i,2}\colon\ \mathit{last} := i; \\
& & \quad en_{i,3}\colon\ \textbf{await } \left(\neg\, \mathit{req}_{i\oplus 1} \vee \mathit{last} \neq i\right); \\
& & \quad cr_i\colon\ C_{i,cr}; \\
& & \quad ex_i\colon\ \mathit{req}_i := \mathit{ff} \\
& & \textbf{end}
\end{array}
$$

If the variable *last* is updated *after* req_i is set, we obtain a correct solution. Just like with $C_{i,4}$, all three properties are satisfied.

If, however, *last* is updated *before* req_i, the resulting protocol is incorrect.

$$
\begin{array}{lcl}
C''_{i,4} & = & \textbf{while } \textit{true} \textbf{ do} \\
 & & \quad nc_i:\ C_{i,nc}; \\
 & & \quad en_{i,1}:\ \textit{last} := i; \\
 & & \quad en_{i,2}:\ req_i := tt; \\
 & & \quad en_{i,3}:\ \textbf{await}\ \left(\neg\, req_{i\oplus 1} \vee \textit{last} \neq i\right); \\
 & & \quad cr_i:\ C_{i,cr}; \\
 & & \quad ex_i:\ req_i := \textit{ff} \\
 & & \textbf{end}
\end{array}
$$

Exercise 1.2.2 *Which of the three properties would the system using $C''_{i,4}$ violate? Explain your answer by giving a counter example, that is, describe a violating computation path.*

Exercise 1.2.3 *The correctness of the above protocol is subject to the assumption that the critical sections of all processes always terminate. Why?*

1.3 The CTL model checking algorithm

When defining the algorithm below we make use of the fact that the connectives *ff*, $\neg$, $\wedge$, **AF**, **EU**, and **EX** form an adequate set (Theorem 1). So, here's the algorithm:

Input: An interpreted FSA $M = (S, S_0, L, \delta, F)$ and a CTL formula φ

Output: "Yes", if $(M, s_0) \models \varphi$ for all initial states $s_0 \in S_0$. "No", otherwise.

1. **Preprocessing** Translate φ into an equivalent formula φ' that contains only the adequate connectives mentioned in Theorem 1.

2. **Labeling** We compute the set of states that satisfy φ'. Label the states of M with the subformulas of φ' that are satisfied there, starting with the smallest subformulas. Suppose that ψ is a subformula of φ' and that the states satisfying all the immediate subformulas of φ' have already been labeled. The states to label with ψ are determined with the following case analysis. If ψ is

   - *ff*: No states are labeled with *ff*,
   - p: Label state s with p if $eval(p, s) = true$,
   - $\psi_1 \wedge \psi_2$: Label state s with $\psi_1 \wedge \psi_2$ if s is already labeled with ψ_1 and ψ_2,
   - $\neg\,\psi_1$: Label state s with $\neg\,\psi_1$ if s is not already labeled with ψ_1,
   - **EX** ψ_1: Label state s with **EX** ψ_1 if at least one of its successor states is labeled with ψ_1,

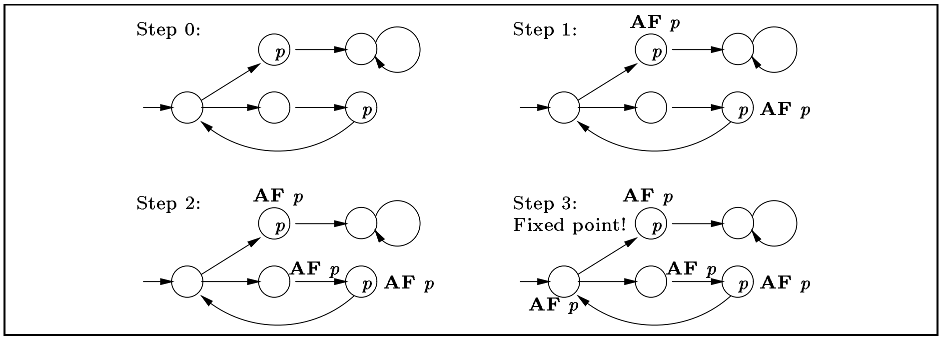


Figure 1.5: Sample execution of the CTL model checking algorithm

- **AF** ψ_1:
  (a) If any state s is already labeled with ψ_1, then label it with **AF** ψ_1,
  (b) label any state with **AF** ψ_1 if all successor states are labeled with **AF** ψ_1,
  (c) if step 2 changed the labeling, then go back to 2. Otherwise, stop.
- $\mathbf{E}[\psi_1 \ \mathbf{U} \ \psi_2]$:
  (a) If any state s is already labeled with ψ_2, then label it with $\mathbf{E}[\psi_1 \ \mathbf{U} \ \psi_2]$,
  (b) label any state with $\mathbf{E}[\psi_1 \ \mathbf{U} \ \psi_2]$ if it is labeled with ψ_1 and at least one of its successor states is already labeled with $\mathbf{E}[\psi_1 \ \mathbf{U} \ \psi_2]$,
  (c) if step 2 changed the labeling, then go back to 2. Otherwise, stop.

3. **Postprocessing** If all initial states S_0 are labeled with φ', output "Yes". Otherwise, output "No".

Figure 1.5 contains a sample execution of the algorithm. Let M be the top left state machine and let φ be the CTL formula **AF** p. The algorithm reaches a fixed point after three steps. Since all initial states are labeled with φ upon termination, the state machine satisfies φ, that is, $M \models \varphi$.

We now present the above algorithm in more concrete terms. We restrict our attention to the labeling step of the algorithm. Figure 1.6 contains the main function whereas Figure 1.7 contains helper functions to handle the cases **EX**, **AF**, and **EU**.

Let us analyze the complexity of this algorithm. The clauses for **AF** and **EU** are the most expensive. In particular, they both contain a loop which, in every iteration, applies a labeling step to every vertex in the graph and which terminates only when these labeling steps stop incurring any changes. Using a standard breadth-first traversal algorithm, every node in a graph can be visited in $O(\mid S \mid + \mid R \mid)$ giving the worst-case complexity of a single

function $\mathrm{SAT}(\varphi)$ =
(* returns the set of states satisfying φ *)
begin **case** φ **of**
  tt : **return** S
  ff : **return** $\emptyset$
  p : **return** $\{s \mid p \in L(s)\}$
  $\neg\, \varphi_1$: **return** $S \backslash \mathrm{SAT}(\varphi_1)$
  $\varphi_1 \wedge \varphi_2$: **return** $\mathrm{SAT}(\varphi_1) \cap \mathrm{SAT}(\varphi_2)$
  $\varphi_1 \vee \varphi_2$: **return** $\mathrm{SAT}(\varphi_1) \cup \mathrm{SAT}(\varphi_2)$
  $\varphi_1 \rightarrow \varphi_2$: **return** $\mathrm{SAT}(\neg\, \varphi_1 \vee \varphi_2)$
  $\mathbf{AX}\ \varphi_1$: **return** $\mathrm{SAT}(\neg\, \mathbf{EX}\ \neg\, \varphi_1)$
  $\mathbf{EX}\ \varphi_1$: **return** $\mathrm{SAT}_{\mathbf{EX}}(\varphi_1)$
  $\mathbf{AF}\ \varphi_1$: **return** $\mathrm{SAT}_{\mathbf{AF}}(\varphi_1)$
  $\mathbf{EF}\ \varphi_1$: **return** $\mathrm{SAT}(\mathbf{E}[tt\ \mathbf{U}\ \varphi_1])$
  $\mathbf{AG}\ \varphi_1$: **return** $\mathrm{SAT}(\neg\, \mathbf{EF} \neg\, \varphi_1)$
  $\mathbf{EG}\ \varphi_1$: **return** $\mathrm{SAT}(\neg\, \mathbf{AF} \neg\, \varphi_1)$
  $\mathbf{A}[\varphi_1\ \mathbf{U}\ \varphi_2]$: **return** $\mathrm{SAT}(\neg\, (\mathbf{E}[\neg\, \varphi_2\ \mathbf{U}\ (\neg\, \varphi_1 \wedge \neg\, \varphi_2)] \vee \mathbf{EG} \neg\, \varphi_2))$
  $\mathbf{E}[\varphi_1\ \mathbf{U}\ \varphi_2]$: **return** $\mathrm{SAT}_{\mathbf{EU}}(\varphi_1, \varphi_2)$
 end
end

Figure 1.6: The function SAT. Given a CTL formula φ it returns the set of states satisfying φ. Uses helper functions in Figure 1.7.

function $\mathrm{SAT}_{\mathbf{EX}}(\varphi)$ =
(* returns the set of states satisfying $\mathbf{EX}\ \varphi$ *)
new X, Y;
begin
  $X :=\mathrm{SAT}(\varphi)$;
  $Y :=\{s_0 \in S \mid R(s_0, s_1) \text{ for some } s_1 \in X\}$
  return Y
end

function $\mathrm{SAT}_{\mathbf{AF}}(\varphi)$ =
(* returns the set of states satisfying $\mathbf{AF}\ \varphi$ *)
new X, Y;
begin
  $X := \emptyset$; $Y :=\mathrm{SAT}(\varphi)$;
  while $X \neq Y$ **do**
    $X := Y$;
    $Y := Y \cup \{s \mid \text{ for all } s' \text{ with } R(s, s') \text{ we have } s' \in Y\}$
  end;
  return Y
end

function $\mathrm{SAT}_{\mathbf{EU}}(\varphi, \psi)$ =
(* returns the set of states satisfying $\mathbf{E}[\varphi\ \mathbf{U}\ \psi]$ *)
new W, X, Y;
begin
  $W :=\mathrm{SAT}(\varphi)$; $X := S$; $Y :=\mathrm{SAT}(\psi)$;
  while $X \neq Y$ **do**
    $X := Y$;
    $Y := Y \cup \big(W \cap \{s \mid \text{ exists } s' \text{ with } R(s, s') \text{ and } s' \in Y\}\big)$
  end;
  return Y
end

Figure 1.7: Helper functions for function SAT in Figure 1.6.

execution of step 2 where $| S |$ and $| R |$ denote the size of the state space and the transition relation respectively. In the worst case, step 2 is executed $| S |$ times. Thus, the complexity of each individual clause of the algorithm is $O(| S | \cdot (| S | + | R |))$. The number of subformulas of a formula is linear in the number n of connectives in that formula. Thus, the complexity of the entire algorithm is $O(n \cdot | S | \cdot (| S | + | R |))$. Since we want to be able to verify large systems with lots of states, the above result is not encouraging.

1.3.1 Optimizations

Explicit treatment of AX, EF, AG, EG and AU

The labeling algorithm treats the connectives **AX**, **EF**, **AG**, **EG** and **AU** in terms of the adequate connectives **EX**, **AF**, and **EU**. This means that the labeling algorithm only has to handle three different cases and allows a very concise presentation. However, it also slows the algorithm down by a constant factor. To handle $\mathbf{AX}\varphi$, for instance, we have to compute all states satisfying φ, $\neg\ \varphi$, $\mathbf{EX}\neg\ \varphi$, and then finally, $\neg\ \mathbf{EX}\neg\ \varphi$. An explicit treatment of **EX** computes the states satisfying $\mathbf{AX}\varphi$ directly from those satisfying φ:

- $\mathbf{AX}\ \psi_1$: Label state s with $\mathbf{AX}\ \psi_1$ if all of its successor states are labeled with ψ_1,

Similarly for **EF** and **AU**:

- $\mathbf{EF}\ \psi_1$:
  1. If any state s is already labeled with ψ_1, then label it with $\mathbf{EF}\ \psi_1$,
  2. label any state with $\mathbf{EF}\ \psi_1$ if some successor state is labeled with $\mathbf{EF}\ \psi_1$,
  3. if step 2 changed the labeling, then go back to 2. Otherwise, stop.
- $\mathbf{A}[\psi_1\ \mathbf{U}\ \psi_2]$:
  1. If any state s is already labeled with ψ_2, then label it with $\mathbf{A}[\psi_1\ \mathbf{U}\ \psi_2]$,
  2. label any state with $\mathbf{A}[\psi_1\ \mathbf{U}\ \psi_2]$ if it is labeled with ψ_1 and all of its successor states are already labeled with $\mathbf{A}[\psi_1\ \mathbf{U}\ \psi_2]$,
  3. if step 2 changed the labeling, then go back to 2. Otherwise, stop.

The explicit treatment of **AG** is slightly different:

- $\mathbf{AG}\ \psi_1$:
  1. Label all states with $\mathbf{AG}\ \psi_1$,
  2. remove label $\mathbf{AG}\ \psi_1$ from any state s, if s is not labeled ψ_1,
  3. remove label $\mathbf{AG}\ \psi_1$ from any state s, if s at least one successor not labeled with $\mathbf{AG}\ \psi_1$,

4. if step 3 changed the labeling, then go back to 3. Otherwise, stop.

The explicit treatment of **EF** is similar. Adding these cases to the algorithm speeds it up by a constant factor.

Bibliography

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[Bah99] A. Bahrami. *Object-Oriented Systems Development.* McGraw-Hill, 1999.

[BBFM99] P. Behm, P. Benoit, A. Faivre, and J.M. Meynadier. Météor: A successful application of B in a large project. In *Proceedings of the World Conference on Formal Methods in the Development of Computing Systems*, pages 369–387. Springer Verlag, 1999. LNCS 1708.

[BEKV94] K. Broda, S. Eisenbach, H. Khoshnevisan, and S. Vickers. *Reasoned Programming.* Prentice Hall International, 1994.

[Boo54] G. Boole. *An Investigation of the Laws of Thought.* Dover, New York, 1854.

[Boo91] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, Menlo Park, CA, 1991.

[Bow] J. Bowen. World wide web virtual library: The Z method. `http://www.afm.sbu.ac.uk/z`.

[Cam] Cambridge University and Technical University of Munich. *Isabelle: A generic theorem proving environment.* http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[CE81] E.M. Clarke and A. Emerson. Synthesis of synchronous skeletons for branching time temporal logic. In D. Kozen, editor, *Logic of Programs Workshop*. Springer Verlag, 1981. LNCS 131.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[CM90] R.H. Cobb and H.D. Mills. Engineering software under statistical quality control. *IEEE Software*, (6):45–54, 1990.

[Com99] President's Information Technology Advisory Committee. Information technology research: Investing in our future. report to the u.s. president. Technical report, National Coordination Office for Information Technology Research and Development, February 1999. Available at `http://www.hpcc.gov/pitac/report/`.

[CY89] P. Coad and E. Yourdon. *OOA-Object-Oriented Analysis*. Englewood Cliffs New Jersey: Prentice Hall, 1989.

[DBK03] E. Kamstries D.M. Berry and M.M. Krieger. From contract drafting to software specification: Linguistic sources of ambiguity. Available at `se.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf`, November 2003.

[DHH01] M. Dwyer, J. Hatcliff, and R. Howell. Slides for course on Software Specifications (CIS771), Kansas State University. Personal communication, November 2001.

[Fre03] G. Frege. Grundgesetze der Arithmetik, begriffsschriftlich abgeleitet, 1903. Volumes I and II (Jena).

[FS97] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modelling Language*. Addison Wesley, 1997.

[Gat02] W. Gates. Trustworthy computing. Email to Microsoft employees sent on January 15, 2002. Available at `http://www.wired.com/news/business/0,1367,49826,00.html`, 2002.

[Ge93] John V. Guttag and James J. Horning (editors). *Larch: Languages and Tools for Formal Specification*. Springer Verlag, 1993.

[Gen69] G. Gentzen. Investigations into logical deduction. In M.E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–129. North Holland, 1969.

[GM93] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gro95a] The Standish Group. Chaos. Technical Report T23E-T10E, The Standish Group, 1995.

[Gro95b] The Standish Group. Chaos. Technical Report T23E-T10E, The Standish Group International, INC., 1995.

[Gro97] Object Modelling Group. Object constraint language specification. Technical report, OMG, September 1997. `ftp://ftp.omg.org/pub/docs/ad/97-08-08.pdf`.

[HR00] M. Huth and M. Ryan. *Logic in Computer Science: Modeling and reasoning about systems.* Cambridge University Press, 2000.

[Jac98] M. Jackson. *Software Requirements & Specifications.* Addison Wesley, 1998.

[Jac99] D. Jackson. A Comparison of Object Modelling Notations: Alloy, UML and Z. MIT, Lab for Computer Science, August 1999.

[Jac00] D. Jackson. Lecture notes for 6.170 (Laboratory in Software Engineering). MIT Laboratory for Computer Science, October 2000.

[Jac01] D. Jackson. Micromodels of software: Modelling & analysis with alloy. Draft, available at `http://sdg.lcs.mit.edu/alloy/book.pdf`, November 2001.

[Jac06] M. Jackson. *Software Abstractions: Logic, Languge, and Analysis.* MIT Press, 2006.

[JCJG92] I. Jacobson, M. Christerson, P. Jonsson, and O. Gunnar. *Object-oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley, 1992.

[Jon86] C.B. Jones. *Systematic Software Development Using VDM.* Prentise Hall, 1986.

[KMM00a] M. Kaufmann, P. Manolios, and J. Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, June 2000.

[KMM00b] M. Kaufmann, P. Manolios, and J. Strother Moore, editors. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[LG00] B. Liskov and J. Guttag. *Program Development in Java.* Addison Wesley, 2000.

[Mar04] J. Marks. Keynote at sigsoft/fse '04. Newport Beach, California, November 2004.

[McC93] S. McConnell. *Code Complete.* Microsoft Press, 1993.

[MDL90] H.D. Mills, M. Dyer, and R.C. Linger. Cleanroom software engineering. *IEEE Software*, pages 19–25, September 1990.

[Mey85] B. Meyer. On formalism in specifications. *IEEE Software*, 3(1):6–25, 1985.

[MMZ01] *Chaff: engineering an efficient SAT solver*, 2001. Las Vegas, Nevada, United States.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety*. Springer, 1995.

[oCP03] ACM Committee on Computers and Public Policy. The risks digest: A forum on risks to the public in computers and related systems, 2003. Moderated by Peter Neumann. Available at `http://catless.ncl.ac.uk/Risks`.

[ORA] ORA Canada. *Never: An automated deduction system.* http://www.ora.on.ca/eves.html.

[Oxf] Oxford University. *Just Another Proof Editor (JAPE).* http://users.comlab.ox.ac.uk/bernard.sufrin/jape.html.

[Pet96] I. Peterson. *Fatal Defect: Chasing Killer Computer Bugs*. Vintage Books, 1996.

[Pra65] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almquist & Wiksell, 1965.

[Pre05] R. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 2005. Sixth Edition.

[PS99] R. Pooley and P. Stevens. *Using UML. Software Engineering with Objects and Components*. Addison Wesley, 1999.

[QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium on Programming*, pages 337–350, 1981.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modelling and Design*. Prentise Hall, 1991.

[Res03] Microsoft Research. Projects. Web page at `http://research.microsoft.com/research/projects`, 2003.

[RG00] M. Richters and M. Gogolla. Validating UML models and OCL constraints. In Springer Verlag, editor, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference*, LNCS 1939, 2000.

[SMF03] A.N. Clark S.J. Mellor and T. Futagami. Model-driven development. *IEEE Software*, (13), September/October 2003.

[Som04] I. Sommerville. *Software Engineering*. Addison-Wesley, 2004. Seventh edition.

[Spi89] J.M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 1989.

[SRI] SRI International. Computer Science Laboratory. *Prototype Verification System (PVS)*. http://pvs.csl.sri.com.

[vEVD89] P.H.J. van Eijk, C. A. Vissers, and M. Diaz. *The formal description technique LOTOS*. Elsevier Science Publishers B.V., 1989.

[Win95] J. Wing. Hints to specifiers. Technical Report CMU-CS-95-118R, CMU, 1995. Available at www.cs.cmu.edu/ wing.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language. Precise Modeling with UML.* Object Technology Series. Addison Wesley, 1999.

[Zha97] *SATO: An efficient propositional prover*, July 1997.