# Busy Wait Analysis

Johann Blieberger[1], Bernd Burgstaller[1], and Bernhard Scholz[2]

[1] Institute for Computer-Aided Automation, TU Vienna
Treitlstr. 1–3, A-1040 Vienna, Austria
{blieb,bburg}@auto.tuwien.ac.at

[2] Institute of Computer Languages, TU Vienna
Argentinierstr. 8/4, A-1040 Vienna, Austria
scholz@complang.tuwien.ac.at

**Abstract.** A busy wait loop is a loop which repeatedly checks whether an event occurs. Busy wait loops for process synchronization and communication are considered bad practice because (1) system failures may occur due to race conditions and (2) system resources are wasted by busy wait loops. In general finding a busy wait loop is an undecidable problem. To get a handle on the problem, we introduce a decidable predicate for loops that will spot most important classes of busy waiting although false alarms may occur. The loop predicate for detecting busy wait loops is based on control flow graph properties (such as loops) and program analysis techniques.

## 1 Introduction

Although for efficiency reasons busy waiting is employed in operating system kernels for process synchronization ("spin locks"), it is considered bad practice to use it for task synchronization and task communication in application programs. Busy waiting results in a waste of system resources. Programs, that actively wait for an event, may cause a severe overhead in a multi-tasking environment and can cause system failure due to race conditions. Therefore, programmers should use higher communication facilities such as semaphores, monitors, rendezvous, etc. [Hoa85,Ada95].

However, it is hard to detect busy waiting in existent code and therefore it is of great importance to have a static analysis tool that targets the detection of busy waiting. Such a tool significantly improves the quality of software in order to prevent programs that use busy waiting.

Before we discuss busy waiting, we have to note that this term is extremely vague. In fact, there is no definition of busy waiting, everybody agrees upon. For example in [And91] it is defined as

> *"... a form of synchronization in which a process repeatedly checks a condition until it becomes true... ".*

We start by defining what we mean by busy waiting: A program that within a loop constantly reads a value from a certain variable, where the loop exit condition is dependent on this value, imposes busy waiting and we say that the loop is a *busy wait loop*. We assume that only another thread/task can terminate the loop by altering the value of the variable. A variable being responsible for busy waiting is called a *wait variable*.

Since busy wait loops may loop forever, the general problem of spotting them is equivalent to the halting problem and thus undecidable. Hence we cannot find all busy wait loops automatically. On the other hand, our analysis will raise false alarms in certain cases (e.g. with *blocking assignments* described below). However, we believe that our analysis will find a large class of busy wait loops and even false alarms may stimulate the programmer to improve the program code.

The paper is structured as follows. In Section 2 we give definitions used throughout the paper. In Section 3 we motivate our analysis. In Section 4 we describe the algorithm for detecting busy wait loops. Finally we draw our conclusions in Section 6.

## 2 Background

A control flow graph $G < N, E, e, x >$ is a directed graph [ASU86] with node set $N$ and edge set $E \subseteq N \times N$. Nodes $n \in N$ represent basic blocks consisting of a linear sequence of statements. Edges $(u, v) \in E$ represent the non-deterministic branching structure of $G$, and $e$ and $x$ denote the unique *start* and *end node* of $G$, respectively. Moreover, $succ(u) = \{v \mid (u, v) \in E\}$ and $pred(u) = \{v \mid (v, u) \in E\}$ represent the *immediate successor* and *predecessor* of node $u$. A *finite path* of $G$ is a sequence $\pi =< u_1, u_2, \ldots, u_k >$ of nodes such that $u_{i+1} \in succ(u_i)$ for all $1 \leq i < k$. Symbol $\varepsilon$ denotes the empty path.

A path $\pi =< u_1, \ldots, u_k >$ is said to be a member of a node set $X$ ($\pi \in X$), if all nodes in the path are members of $X$.

Let node $u$ *dominate* [Muc00] node $v$, written $u \operatorname{dom} v$, if every possible path from the start node $e$ to node $v$ includes $u$. The domination relation $u \operatorname{dom} v$ is reflexive ($u \operatorname{dom} u$), transitive ($u \operatorname{dom} v \wedge v \operatorname{dom} w \Rightarrow u \operatorname{dom} w$), and anti-symmetric ($u \operatorname{dom} v \wedge v \operatorname{dom} u \Rightarrow u = v$).

For every node $u \in N \setminus \{e\}$ there exists an immediate dominator $v$, written as $v = \operatorname{idom}(u)$ such that there exists no dominator $w \neq v$ of $u$ which is dominated by $v$. The immediate dominators construct a tree also known as the dominator tree. The dominator tree is a compressed representation of the domination relation.

A *back edge* $(m, n) \in E$ in a control flow graph $G$ is defined to be an edge whose target dominates its source. The set of back edges is defined to be $B = \{(m, n) \in E \mid n \operatorname{dom} m\}$.

## 3 Motivation

Our intuition of a busy wait loop is based on the notion of loops and the read/write semantics of program variables inside the loop. According to [ASU86]

```
1   Turn : Integer := 1;
2   Flag0 : Boolean := False;
3   Flag1 : Boolean := False;

4   procedure P0 is
5   begin
6     -- Claim Critical Section:
7     Flag0 := True;                        -- Node 1
8     while Flag1 = True loop               -- Node 2
9        if Turn = 1 then                   -- Node 3
10          Flag0 := False;                 -- Node 4
11          while Turn = 1 loop             -- Node 5
12                null;                     -- Node 6
13          end loop;
14          Flag0 := True;                  -- Node 7
15       end if;
16    end loop;
17    -- Critical Section:
18    null;                                 -- Node 8
19    -- Leave Critical Section:
20    Turn := 1;                            -- Node 8
21    Flag0 := False;                       -- Node 8
22  end P0;
```

**Fig. 1.** Running Example: Dekker's Algorithm

a *definition* of a variable $x$ is a statement that assigns $x$ a value. Contrary, a variable *declaration* is a syntactic constructs which associates information (e.g. type information) with a given name. A variable declaration usually implies also an initial definition for the variable itself.

With our analysis we are interested in program variables which determine whether the loop is terminated or iterated again. We assume that we find these program variables in the exit-condition of loops. If such a variable is only read, without employing higher communication facilities or being *defined* inside the loop, this variable might be responsible for inducing busy waiting and we call this variable a *wait variable*.

We illustrate these terms by the mutual exclusion algorithm given in Figure 1. Dijkstra [Dij68] attributes this algorithm to the Dutch mathematician T. Dekker, and it is in fact the first known solution to the mutual exclusion problem for two processes that does not require strict alternation. It is worth noting that this algorithm only assumes mutual exclusion at the memory access level which means that simultaneous memory access is serialized by a memory arbiter. Beyond this, no further support from the hardware (e.g. atomic test and set instructions), operating system, or programming language is required. Since, for this reason, the algorithm solely relies on global variables (cf. lines $1 \ldots 3$) that are written and read for synchronization purposes, we have found it to be an instructive example of busy waiting behavior. Note that we have omitted the code for the second process (P1), as it is dual to P0.

In order to demonstrate the overhead in CPU processing time induced by busy waiting we have implemented Dijkstra's $N$-way generalization [Dij65] of Dekker's 2-way mutual exclusion algorithm for the RTAI [M$^+$00] real-time Linux executive run on a uni-processor platform. Our investigation focused on the average execution time of a busy waiting real-time task that has to enter the critical section a constant number of times in the presence of $N-1$ busy waiting competitors with the same assignment. In Figure 2 these results are compared to an implementation utilizing a semaphore to ensure mutual exclusion between tasks. Due to the blocking nature of the semaphore the measured execution times show only linear growth in terms of an increasing number of participating tasks. This behavior is clearly exceeded by the busy waiting task ensemble that spends most of its execution time polling to gain access to the critical section.

It is clear that our example contains busy waiting and our objective is to design an algorithm that detects this busy waiting behavior just by inspecting the loops of the program and the read/write semantics of program variables inside the loop.

*Non-dangerous statements* are statements which prevent a variable from being a wait variable. We call non-dangerous assignment statements *blocking assignments.* If a variable is defined by some of these statements, we assume that busy waiting is improbable to occur. Statements which are not non-dangerous are called *dangerous*. We discriminate between tasking statements of Ada and other statements.

1. All calls to a (guarded[3]) entry of a task or protected object are considered non-dangerous.
2. Timed entry calls are non-dangerous iff the expiration time does not equal zero.
3. Asynchronous transfer of control (ATC) is non-dangerous if the triggering statement is an entry call (to a guarded entry).
4. A timeout realized by ATC is considered non-dangerous.
5. Conditional entry calls are generally considered dangerous.
6. In general we assume that file operations are non-dangerous; the same applies to Terminal I/O. We do not consider cases such as a file actually being a named pipe which is fed by a program providing output in an infinite loop.
7. Read/Write attributes may or may not block depending on the actual implementation. For this reason we consider read/write attributes dangerous.
8. Assignments via function or procedure calls are dangerous even if inside the subprogram there is a blocking assignment (we do not perform interprocedural analysis).

[3] A call to a guarded entry is a non-dangerous statement with high probability except if the guard equals *true*; a call to a task entry without a guard has high probability to be dangerous, except if the corresponding accept statement is located in a part different from the main "select" loop in the task body (which makes sense for task initialization and finalization).

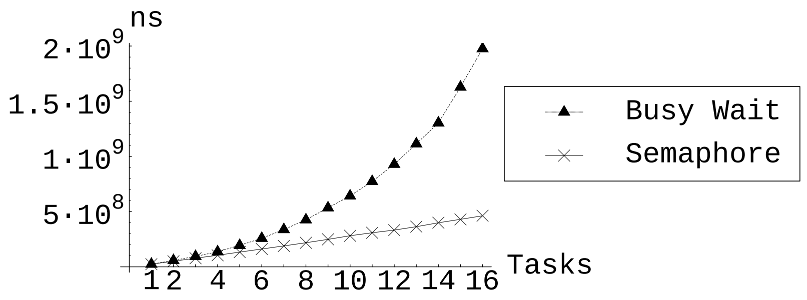


**Fig. 2.** Task Execution Times: Busy Waiting vs. High-Level Synchronization

## 4 Algorithm

For detecting busy wait loops we analyze loops of a program. We use the control flow graph as an underlying data structure for the detection. Based on control flow properties and semantic properties of statements, we decide whether a loop is a busy wait loop or not.

In general it is not easy to find loops in control flow graphs [Ram99]. However, if those graphs are *reducible* (cf. [ASU86]), loops are simple to define [Muc00]. In the following we introduce *natural loops*, which are defined by their back-edges[4].

A *natural loop* of a back-edge $(m, n)$ is the sub-graph consisting of the set of nodes containing $n$ and all the nodes which can reach $m$ without passing $n$. Let $L_{(m,n)}$ denote the set of nodes which induce the sub-graph that forms the natural loop of back-edge $(m, n)$. Then,

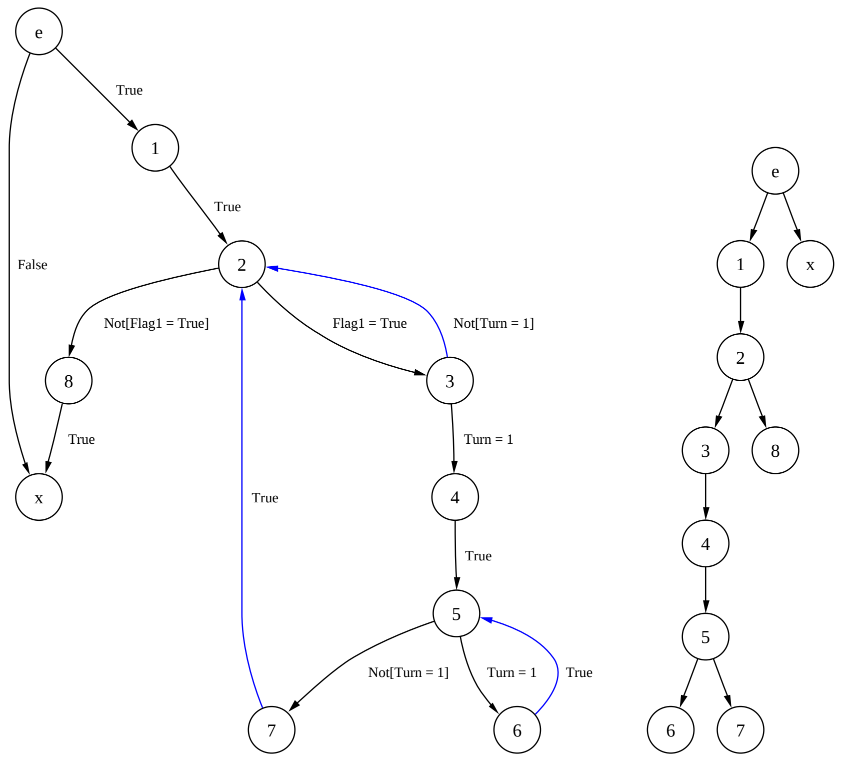
$$L_{(m,n)} = \{u \mid \exists \pi =< u, \ldots, m >: n \notin \pi\} \cup \{n\}. \tag{1}$$

Node $n$ is said to be the *loop header* of loop $L_{(m,n)}$ because the loop is entered through $n$. Therefore, the loop header dominates all nodes in the loop.

The algorithm for computing $L_{(m,n)}$ can be found in Figure 4. It is a simple work-list algorithm. It computes the immediate and intermediate successors of node $m$. The algorithm stops after all successors of $m$ excluding the successors of $n$ have been found. The immediate and intermediate successors excluding the successor of $n$ represent the set of nodes for the loop.

Recall our example of Figure 1. The control flow graph of our running example is given in Figure 3(a). In addition the dominator tree is depicted in Figure 3(b). To compute the dominator tree several algorithms have been pro-

[4] Note that this definition is only valid for reducible control flow graphs. However, Ada programs result in reducible control flow graphs only, and this is no restriction for the analysis.



(a) Control Flow Graph (b) Dominator Tree

**Fig. 3.** Running Example: Dekker's Algorithm

posed in literature [Muc00,LT79,AHLT99]. The complexity of the algorithms varies from cubic to linear[5].

For finding the loops of our example we determine the back edges occurring in the control flow graph of Figure 3(a). The set $B$ of back edges consists of the following edges:

$$B = \{(3,2),(6,5),(7,2)\}. \tag{2}$$

Based on the set of back edges we compute the set of nodes of each natural loop as given in the algorithm shown in Figure 4.

| $(m,n)$ | $L_{(m,n)}$ |
|---|---|
| $(3,2)$ | $\{2,3\}$ |
| $(6,5)$ | $\{5,6\}$ |
| $(7,2)$ | $\{2,3,4,5,6,7\}$ |

[5] It seems to be the case that if a lower complexity is desired more effort has to be put into implementing the algorithm.

```
1: $W := \{m\}$;
2: $L_{(m,n)} := \{n\}$;
3: **repeat**
4:    *select* $u \in W$;
5:    $L_{(m,n)} := L_{(m,n)} \cup \{u\}$;
6:    $W := (W \cup succ(u)) \setminus L_{(m,n)})$;
7: **until** $W = \emptyset$
```

**Fig. 4.** Algorithm for Computing $L_{(m,n)}$.

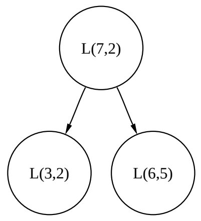


**Fig. 5.** Loop Forest of Running Example.

For detecting busy waiting behavior of a program we are interested in the innermost loops containing one or more wait variables to focus the programmer on the smallest portion of code which may induce a *busy wait loop*. For finding the innermost loops we construct a loop forest [Ram99]. Nodes in the forest represent loops and an edge denotes a nesting relation. For our example the loop forest is depicted in Figure 5. It shows that loop $L_{(7,2)}$ contains the two inner loops, i.e., $L_{(6,5)}$ and $L_{(3,2)}$. Note that the loop forest represents the set-relationship $L_1 \subset L_2$ between two loops $L_1$ and $L_2$. In the example the following holds: $L_{(6,5)} \subset L_{(7,2)}$ and $L_{(3,2)} \subset L_{(7,2)}$.

For locating the busy wait loop we will analyze the loops in reverse topological order of the loop forest, i.e. from the inner-most loops to the top-level loops of a program. The reverse topological order guarantees that the busy wait behavior of the program can be localized quite precisely.

Now, we want to deduce the set of statements, which influence the termination of a loop. These are statements inside the loop, which have at least one immediate successor that is outside the loop. For example an `exit when ...` statement might terminate the loop.

The statements that influence the termination of the loop are given by the following set of edges:

$$T_{(m,n)} = \{(u,v) \in E \mid u \in L_{(m,n)} \wedge v \notin L_{(m,n)}\} \tag{3}$$

The definition establishes those edges whose sources but not its destinations are part of the loop $L_{(m,n)}$. Note that a statement in loop $L_{(m,n)}$ must have at least two successors for its out-going edges to contribute to the set $T_{(m,n)}$. Therefore, the statement must be a branching node and there must be a branch predicate that decides whether to stay in the loop or to exit the loop. The branch predicate consists of program variables and we call these variables candidates for *wait variables.* These wait variables might cause a busy wait loop and must be checked.

The set of candidates, which might be wait variables, are given as follows,

$$V_{(m,n)} = \{\mathbf{var} \in bp(u) \mid (u,v) \in T_{(m,n)}\} \tag{4}$$

where $bp(u)$ denotes the branch predicate of branching node $u$ and **var** is a candidate for a wait variable and hence needs to be added to the set $V_{(m,n)}$. For our example given in Figure 1 the T-sets and V-sets are as follows:

| $(m,n)$ | $T_{(m,n)}$ | $V_{(m,n)}$ |
|---|---|---|
| $(3,2)$ | $\{(3,4),(2,8)\}$ | $\{\text{Turn}, \text{Flag1}\}$ |
| $(6,5)$ | $\{(5,7)\}$ | $\{\text{Turn}\}$ |
| $(7,2)$ | $\{(2,8)\}$ | $\{\text{Flag1}\}$ |

In the example we have three branching nodes which might cause a busy wait loop, i.e. nodes 2, 3, and 5. For example the branch predicate of branching node 2 is given as `Flag1 = true`. Variable `Flag1` occurs in the branch predicate and, therefore, it is candidate for a busy wait variable, i.e. `Flag1` $\in V_{(3,2)}$ and `Flag1` $\in V_{(7,2)}$.

Now, we have to check if a candidate for a busy wait variable might be a wait variable and might cause the loop to loop forever without any interaction from another thread/task. This might happen if there is a path in the (natural) loop that does not contain a definition for a canditate.

For a basic block $u$ we introduce a local predicate $\mathit{defined}_{\mathbf{var}}(u)$. The predicate holds for a blocking assignment for variable **var** in basic block $u$. For example statement `Flag0:=False;` in node 4 of our running example contains a blocking assignment for variable `Flag0`. Therefore, predicate $\mathit{defined}_{\texttt{Flag0}}(4)$ holds.

We extend the definition of $\mathit{defined}_{\mathbf{var}}$ for paths. If in a path $\pi$ there exists at least one blocking assignment for variable **var**, the predicate $\mathit{defined}_{\mathbf{var}}(\pi)$ holds. The extension is obtained as follows:

$$\mathit{defined}_{\mathbf{var}}(< u_1, u_2, \ldots, u_k >) = \bigvee_{1 \le i \le k} \mathit{defined}_{\mathbf{var}}(u_i) \tag{5}$$

**Definition 1.** *If for a variable* **var** *the* **busy-var**-*predicate*

$$\mathbf{busy\text{-}var}(L_{(m,n)}, \mathbf{var}) =_{\text{defs}} \exists \pi =< n, \ldots, n > \in L_{(m,n)} : \neg \text{defined}_{\mathbf{var}}(\pi) \tag{6}$$

*holds in loop $L_{(m,n)}$, the loop is supposed to be a busy wait loop.*

For each program variable $\mathbf{var} \in V_{(m,n)}$ we construct the induced subgraph of the nodes in the loop where $\mathit{defined}_{\mathbf{var}}(u)$ is false. If $m$ is reachable from $n$ in this induced sub-graph, it is simple to show that the predicate **busy-var**$(L_{(m,n)}, \mathbf{var})$ holds. The check boils down to a reachability check in the induced subgraph of the node set $L_{(m,n)} \setminus \{u \mid \mathit{defined}_{\mathbf{var}}(u)\}$. In graph theory there are very efficient algorithms for doing so [Meh84,Sed88].

If we consider our example, the loop $L_{(3,2)}$ is such a loop. For both variables `Turn` and `Flag1` in $V_{(3,2)}$ there is no blocking assignment in nodes 2 and 3. Therefore, node 3 is reachable from node 2 and the predicate **busy-var**$(L_{(m,n)}, \texttt{Turn})$ and **busy-var**$(L_{(m,n)}, \texttt{Flag1})$ is true. Similar accounts for the other loops and their variables of our running example, which implies that all loops of our running example are busy wait loops.

Finally, we put together our detection algorithm in Figure 6. In the pre-phase we build the dominator tree and determine the back-edges. Then, we compute the loop forest which tells us the loop order of the analysis. In reverse topological order of the loop forest, we check loop by loop. For each loop we compute the set of nodes which terminate the loop ($T_{(m,n)}$) and the variables inside the loop exit condition ($V_{(m,n)}$). For every variable in $V_{(m,n)}$ we compute the busy wait predicate and if it is true, we output a warning for this particular loop.

```
1:  compute dominator tree
2:  determine back edges
3:  compute loop forest
4:  for $(m, n)$ in reverse topological order of loop forest do
5:      compute $T_{(m,n)}$
6:      compute $V_{(m,n)}$
7:      for var $\in V_{(m,n)}$ do
8:          determine busy-var$(L_{(m,n)}, \mathbf{var})$
9:          if busy-var$(L_{(m,n)}, \mathbf{var})$ then
10:             output warning for var and loop $L_{(m,n)}$
11:         end if
12:     end for
13: end for
```

**Fig. 6.** Algorithm for Detecting Busy Waiting

## 5 Refinement

By a slight modification we can improve (sharpen) our analysis, i.e., we can detect more busy wait loops.

Consider the code fragment given in Figure 7. In this case our algorithm finds an assignment to the variable $i$ within the loop body and concludes that there is no busy waiting, which is plainly wrong.

This behavior can be improved by considering all variables appearing on the right hand side of assignments to candidate variables to be candidates as well.

```
i,j: integer := 0;

loop
  i := j;
  exit when i=1;
end loop;
```

**Fig. 7.** Example: Indirect Busy Wait Variable

Formally we have to redefine $V_{(m,n)}$ in the following way:

$$\begin{aligned} V^0_{(m,n)} &= \{\textbf{var} \in bp(u) \mid (u,v) \in T_{(m,n)}\} \\ V^{k+1}_{(m,n)} &= \{\textbf{var} \in \text{rhs of assignments in } L_{(m,n)} \text{ to } \textbf{var} \in V^k_{(m,n)}\} \\ V_{(m,n)} &= \bigcup_{k \geq 0} V^k_{(m,n)} \end{aligned}$$

This refinement of our analysis sharpens its results in that more busy wait loops are detected, but on the other hand more false alarms can be raised. For example replace the assignment `i := j;` with `i := j+i-j+1;` in Figure 7. In this case busy waiting will be reported by our refined algorithm although this is not true.

## 6 Conclusion and Future Work

We have presented an algorithm for detecting busy waiting that can be used either for program comprehension or for assuring code quality criteria of programs. Specifically, if processes or threads are part of a high level language (as with Ada or Java), the programmer should be aware of synchronization mechanisms. A tool that detects busy waiting is of great importance for saving system resources and making a program more reliable.

Symbolic methods such as those introduced in [CHT79] will certainly improve analysis in that less false alarms will be raised and more busy wait loops can be found. We will consider symbolic busy wait analysis in a forthcoming paper.

### Acknowledgments

One of the authors (JB) wants to thank Manfred Gronau for pointing him to the subject of busy wait loops and the fact of missing computer-aided analysis.

## References

[Ada95] ISO/IEC 8652. *Ada Reference Manual*, 1995.

[AHLT99] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.

[And91] G. R. Andrews. *Concurrent Programming, Principles & Practice.* Benjamin/Cummings, Redwood City, California, 1991.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers.* Addison-Wesley, Reading, Massachusetts, 1986.

[CHT79] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Trans. on Software Engineering*, 5(4):403–417, July 1979.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[Dij68] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, 1985.

[LT79] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flow graph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[M$^+$00] P. Mantegazza et al. *DIAPM RTAI Programming Guide 1.0.* Lineo, Inc., Lindon, Utah 84042, US, 2000. `http://www.rtai.org`.

[Meh84] K. Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms.* Springer-Verlag, Berlin, 1984.

[Muc00] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco, 2000.

[Ram99] G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, March 1999.

[Sed88] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, 2$^{\text{nd}}$ edition, 1988.