DATA-FLOW FRAMEWORKS FOR WORST-CASE EXECUTION TIME ANALYSIS

JOHANN BLIEBERGER

Abstract. The purpose of this paper is to introduce frameworks based on data-flow equations which provide for estimating the worst-case execution time (WCET) of (real-time) programs. These frameworks allow several different WCET analysis techniques, which range from naïve approaches to exact analysis, provided exact knowledge on the program behaviour is available. However, data-flow frameworks can also be used for symbolic analysis based on information derived automatically from the source code of the program.

As a byproduct we show that slightly modified elimination methods can be employed for solving WCET data-flow equations, while iteration algorithms cannot be used for this purpose.

1. Introduction

Data-flow analysis is primarily used by compilers to optimize the performance of the generated code (cf. [ASU86]). Data-flow analysis algorithms are also used to solve problems in verification, debugging, testing, parallelization, vectorization and parallel programming environments. They provide information about a program or environment without executing the code.

A lot of theoretical results on data-flow frameworks has been derived (cf. e.g. [KU76, KU77]) and a large number of algorithms has been developed (cf. e.g. [AC76, GW76, HU77, Sre95, SGL98, Tar81a, Tar81b]). See [MR90, RP86] for an overview.

Worst-Case Execution Time (WCET) analysis does not have such a long-standing tradition (cf. e.g. [CBW96, HS91, ITM90, NP93, Par93, PK89, PS97, Sha89]). Designers of real-time programming languages usually restrict language features in order to make it possible to guarantee time bounds and introduce new language features to let the programmer add extraneous information on the algorithms which cannot be determined from the source code.

Several different approaches to WCET analysis have been pursued. The most important ones are described shortly in the following:

1. In [KS86, HS91] *Real-Time Euclid*, a language for implementing real-time systems, is presented. Real-Time Euclid prohibits the use of recursions and goto-statements. Loops are restricted to time bounded loops and simple for-loops. An algorithm for calculating an upper bound of the WCET of Real-Time Euclid programs is described in [SHH91].

   Although in the meantime it has been proved that more general loop-statements can be used in real-time programming languages (cf. [Bli94]) and that recursion can be employed without harm in real-time systems (cf. [BL96, Bli00]), the concept of *schedulability analysis*, which is also introduced in [HS91], is still very important for real-time applications.

   One of the minor results of this paper is that goto-statements can be used for implementing real-time systems without prohibiting schedulability analysis, i.e., although gotos are present in a program, its WCET can be determined

Date: October 25, 2000.

Key words and phrases. real-time systems, worst-case execution time, data-flow analysis, symbolic evaluation.

effectively. We would like to note that this result sets theoretical foundations of real-time systems and does not give arguments of whether gotos should be used in programming using high-order programming languages or not.

2. The idea to estimate WCET of programs written in higher-level languages has been introduced in [Sha89]. So-called *schemas* are used to estimate the best and worst-case execution time of statements of higher-level languages and an extension of Hoare logic (cf. [Hoa69]) is employed to prove the timeliness (and correctness) of real-time programs. The method is also able to handle certain real-time language constructs such as delays and time-outs.

   Although Hoare logic is employed, the user has to give constant loop bounds in order to let the compiler determine upper and lower bounds of the number of iterations of a loop.
3. Continuing and extending [Sha89] best and worst-case execution time is estimated by employing static and dynamic program paths analysis in [Par93]. This is done by specifying program paths by *regular expressions.* Since processing this information sometimes requires exponential time, an *interface definition language* is introduced which allows efficient analysis but does not have the expressive power of regular expressions.
4. Determining the execution time of a code segment is also mentioned in [GR91]. Real-time concurrent C uses a tool which originally is based on [MACT89].
5. In [PK89] language constructs have been introduced in order to let the programmer integrate knowledge about the actual behavior of algorithms which cannot be expressed using standard programming language features. These constructs are *scopes*, *markers*, and *loop sequences.* Markers are used to define the number of loop iterations if this number cannot be estimated from the program automatically, e.g., if a general loop is used. Nevertheless all loops are forced to have a constant upper bound.
6. In [PS97] an *integer linear programming* approach (similar to that of [LM95]) is employed, which together with so-called *T-graphs* is used to determine the WCET of real-time programs. A T-graph is similar to a control flow graph (CFG) of a program. In fact, a T-graph is *dual* to its corresponding CFG, which means that the nodes of the T-graph can be mapped to the edges of the CFG and the edges of the T-graph can be mapped to the nodes of the CFG.

   In [PS97] it is proved that the employed method can be used to determine the *exact* timing behavior of a program and not only an upper bound of it. This can also be proved for our approach. In addition our symbolic approach can handle formal and generic parameters (cf. [Ada95]) too.
7. *Partial evaluation* is used in [NP93] to estimate the execution time of programs at compile time. This is done by use of *compile time variables*, i.e., a variable whose value is definitely known at compile time. Taking advantage of these values, programming language constructs can be simplified thereby speeding up the program in most cases.

   This approach does not need to restrict programming language constructs such as *loops*, *recursion*, or *dynamic storage allocation* as long as compile time known values are involved. It can even solve certain simple problems of concurrent programming and synchronization of concurrent processes at compile time.
8. In [CBW96] WCET analysis and program proof are combined for the SPARK Ada subset (cf. [CJM$^+$92]). It is based on a slight generalization of [Tar81b]. The method allows WCET analysis depending on the program's input data, but not in such a general manner as described in this paper. In particular, the *modes* introduced in [CBW96] appear *automatically* in our approach.

```
procedure power(x,n: positive; y: out positive) is
  h : positive;                                  -- Node 1
  l : positive := x;                             -- Node 1
  e : positive := n;                             -- Node 1
begin
  y := 1;                                        -- Node 1
  while e>0 loop                                 -- Node 2
    h := e mod 2;                                -- Node 3
    e := e / 2;                                  -- Node 3
    if h=1 then                                  -- Node 3
      y := y∗l;                                  -- Node 4
    end if;
    l := l∗l;                                    -- Node 5
  end loop;
end power;
```

Figure 1. A Simple Procedure for Raising to a Power

This paper introduces frameworks based on data-flow equations that provide for estimating the WCET of real-time programs. These frameworks allow several different WCET analysis techniques with various precisions, which range from naïve approaches to exact analysis, provided exact knowledge on the program behavior is available. In addition, data-flow frameworks can also be used for symbolic analysis based on information derived automatically from the source code of the program.

In Section 2 we give a short overview of data-flow frameworks and their most important properties. In Section 3 we set up a simple data-flow framework for WCET analysis which includes most of the state-of-the-art WCET analysis techniques. In Section 4 we provide a fully oracle-based data-flow framework for WCET analysis, which can be shown to produce the exact timing behavior of the underlying program. Section 5 is devoted to symbolic evaluation techniques which can be used to replace the oracle from Section 4. Section 6 discusses our results and compares them to other approaches known from literature.

The underlying program representation of our model is the *control flow graph (CFG)*, a directed labeled graph. Its nodes are the program statements, whereas its edges represent transfers of control between *basic blocks*. We assume that the WCET of each basic block is fixed and can be determined at compile time, i.e., we do not consider effects of caching or pipelining. Cache hit/miss prediction with help of symbolic evaluation is presented in [BFS00]; symbolic analysis of pipelining will be treated in a forthcoming paper.

Example 1. We use a running example shown in Figure 1. The corresponding *control flow graph (CFG)* is depicted in Figure 2. The comments added at statements in Figure 1 indicate in which node of the control flow graph they are contained. Edges in the CFG show possible flow of control in procedure **power**. Note that an extraneous edge from node *entry* to node *exit* has been inserted which has no correspondence to the actual data-flow in procedure **power**; it is present to simplify algorithms based on the CFG.

Remark 1.1. Throughout the paper we use the following notational conventions:

1. We write $\lfloor x \rfloor$ to denote the smallest integer greater or equal to x.
2. By $\lceil x \rceil$ we denote the greatest integer smaller or equal to x
3. By $\operatorname{ld} x$ we denote the binary logarithm of x and by $\log x$ the natural logarithm of x.
4. The empty set is denoted by $\emptyset$.

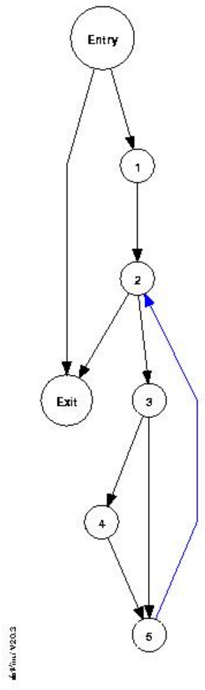

FIGURE 2. Control Flow Graph of Power Example

2. Data-Flow Frameworks

In this section we give a short overview of data-flow frameworks. In most cases we will use the notation and definitions given in [MR90].

In general data-flow analysis algorithms gather facts about the use and definition of data, and information about control and data dependencies in programs. *Data-flow frameworks* are algebraic structures used to encode and solve data-flow problems. A data-flow framework for a problem involves a flow graph, a semilattice of values, and a set of functions from the semilattice to itself. In the following we give definitions and important properties of data-flow frameworks.

Definition 2.1. A data-flow framework D is a quadruple

$$D = \langle G, L, F, M \rangle ,$$

where

$$G = \langle V, E, \rho \rangle$$

is the flow-graph, where V is the set of vertices, E is the set of edges and ρ is the unique entry node and the in-degree of ρ is zero,

$$L = \langle A, \underline{0}, \underline{1}, \wedge \rangle$$

is a meet semilattice, that is

- A is a set (often a power set),
- $\underline{0}$ and $\underline{1}$ are distinguished elements of A,
- $\wedge$ is an operation *meet* with the following properties:

$$\begin{aligned}
&\wedge \text{ is commutative and associative}\\
&a \wedge a = a\\
&a \wedge \underline{0} = \underline{0}\\
&a \wedge \underline{1} = a
\end{aligned}$$

F is a class of functions

$$F \subseteq \{f : L \to L\};$$

- F contains the identity function $\imath$, and
- usually the constant functions $\overline{0}$ and $\overline{1}$, and
- is closed under composition and pointwise meet; that is,

$$\forall f, g \in F : f \circ g \in F$$

(where the notation f^k represents iterated composition of f and $f^0 = \imath$), and

$$\text{if } h(x) = f(x) \wedge g(x), \text{ then } h \in F;$$

and

$$M : E \to F$$

If $p = (p_0, p_1, \ldots, p_n)$ is a path in G with $e_i = (p_{i-1}, p_i)$, then

- $M(\Lambda) = \imath$ where Λ is the empty path and
- $M(p) = M(e_n) \circ M(e_{n-1}) \circ \cdots \circ M(e_1)$.

The meet operation implies a reflexive partial order $\leq$ defined by

$$a \leq b \text{ iff } a \wedge b = a.$$

Example 2. A well-known data-flow problem is the *Reaching Definitions Problem* (cf. e.g. [ASU86, MR90]): A given variable `X` may be defined in several basic blocks of the procedure, and the interesting question is: Which of these definitions reach the entry to some other basic block? This information could be used to answer the questions: Which definitions of `X` reach a given use of `X` in an expression? Is `X` used anywhere before it is defined?

A definition d of `X` in block B is *downward-exposed* if no other definition of `X` occurs after d in B. A downward-exposed definition of X in block B' reaches the entry of block B if there is a path from the exit of B' to B on which no other definition of `X` occurs.

This leads to the following set of equations:

$$\begin{aligned}
\mathrm{Reach}(B) &= \bigcup_{B' \in \mathrm{Preds}(B)} [\mathrm{Reach}(B') \cap \mathrm{Pres}(B') \cup \mathrm{Gen}(B')]\\
\mathrm{Reach}(\rho) &= \emptyset,
\end{aligned}$$

where Reach(B) is the set of definitions reaching the top of B, Preds(B) is the set of *predecessors* of B, Pres(B) is the set of definitions *preserved* through B (that is, not superseded by more recent definitions), and Gen(B) is the set of downward-exposed definitions *generated* in B.

The flow-graph G of the reaching definitions problem is the flow-graph of the underlying procedure. L is the power set lattice on the set of downward-exposed definitions in the procedure (with the reversed order, meet is union, $\underline{1} = \emptyset$ and $\underline{0}$ = the universal set of all downward-exposed definitions). F is the set of functions $\{f(X) = X \cap A \cup B \mid A, B \in L\}$; and M is the mapping assigning to an edge (B, B') the function $f(X) = X \cap \mathrm{Pres}(B) \cup \mathrm{Gen}(B)$.

In the following we will give some properties of data-flow frameworks. Many more and more restrictive properties are discussed in literature. Since, however, our intention is primarily in proving and disproving properties of data-flow frameworks for worst-case execution time analysis, we restrict our interest to the following properties.

Properties 2.1.

1. A semilattice L is *closed under finite meets* if it is closed under arbitrary non-empty meets. If it is also closed under arbitrary infinite meets, it is called *closed under infinite meets* [Kil73].
2. A semilattice has the *descending chain condition (d.c.c.)* if any descending *chain* of semilattice elements
$$x_1 > x_2 > \dots$$
is finite. The relation $>$ is defined by
$$a > b \quad \text{iff } a \geq b \text{ and } a \neq b.$$
3. A function f is *monotone* if
$$\forall f \in F\ \forall x, y \in L : x \leq y \Rightarrow f(x) \leq f(y).$$
f is *distributive* if
$$\forall f \in F\ \forall x, y \in L : f(x \wedge y) = f(x) \wedge f(y),$$
and *continuous for (in)finite meets* if L is closed under arbitrary (non-empty) (in)finite meets and
$$\forall f \in F\ \forall \text{ non-empty sets } \{x_i\}_{i \in I} \subseteq L : f\left(\bigwedge_{i \in I} x_i\right) = \bigwedge_{i \in I} f(x_i).$$
4. For $f \in F$, define
$$f^{[k]} = \bigwedge_{i=0}^{k-1} f^i.$$
We say that F is *bounded* if for any $f \in F$ the chain $\{f^{[i]}\}$ is finite.
5. $f^{@}$ is called a *pseudo-transitive closure* of f if
   (a) $f^{@}(x) \leq f^i(x)\ \forall x \in L,\ i \geq 0$, and
   (b) if $x \in L$ is such that $x \leq f(x)$ then $x \leq f^{@}(x)$.

Example 2. The reaching definition problem is monotone and distributive, closed under infinite meets and continuous for infinite meets, F is bounded* and it has a pseudo-transitive closure.†

The set of equations for the reaching definitions problem of the power example is shown in Table 1 where we have written X_i instead of $\mathrm{Reach}(B_i)$. Note that we can restrict our interest to the variables e, h, l, and y because x and n cannot not be overwritten within procedure **power** (compare [Ada95] for semantic details).

We will use this set of equations in the following to solve the reaching definitions problem by an *iteration algorithm*.

* In fact Reaching Definition has much more restrictive properties but those given above are enough to contrast this example with the data-flow frameworks given in Sections 3, 4, and 5.

† A more restrictive closure exists.

$$\begin{array}{lcl}
X_{\text{entry}} & = & \emptyset \\
X_{\text{exit}} & = & X_2 \cap \{e,h,l,y\} \cup \emptyset \\
X_1 & = & X_{\text{entry}} \cap \{e,h,l,y\} \cup \emptyset \\
X_2 & = & (X_1 \cap \{h,y\} \cup \{e,l\}) \cup (X_5 \cap \{e,h,y\} \cup \{l\}) \\
X_3 & = & X_2 \cap \{e,h,l,y\} \cup \emptyset \\
X_4 & = & X_3 \cap \{l,y\} \cup \{e,h\} \\
X_5 & = & (X_3 \cap \{l,y\} \cup \{e,h\}) \cup (X_4 \cap \{e,h,l\} \cup \{y\})
\end{array}$$

Table 1. Set of Equations for Reaching Definitions Problem of Power Example

X_{entry}	$\{e,h,l,y\}$	$\emptyset$
X_{exit}	$\{e,h,l,y\}$	$\{e,h,l,y\}$
X_1	$\{e,h,l,y\}$	$\emptyset$
X_2	$\{e,h,l,y\}$	$\{e,h,l,y\}$
X_3	$\{e,h,l,y\}$	$\{e,h,l,y\}$
X_4	$\{e,h,l,y\}$	$\{e,h,l,y\}$
X_5	$\{e,h,l,y\}$	$\{e,h,l,y\}$

Table 2. Results of Reaching Definitions Problem of Power Example Obtained by an Iteration Algorithm

1. We first initialize the variables $X_{\text{entry}}, \ldots, X_5$ to $\{e,h,l,y\}$ the zero element ($\underline{0}$) of the underlying semilattice.
2. Then according to the set of equations in Table 1, we produce new values for the variables $X_{\ldots}$ by inserting the old ones on the right side of the equations.
3. This procedure is continued until no change in $X_{\ldots}$ occurs.

We obtain the results in Table 2. As can be seen the algorithm "converges" very fast, which is a great advantage of iteration algorithms. However, as we will see in the following section, they cannot be employed for every data-flow framework.

3. A Data-Flow Framework for Simple WCET

In this section we define a simple data-flow framework for worst-case execution time analysis. The idea is based on methods for WCET analysis widely used in literature, e.g. [PK89, Sha89, HS91, ITM90, GR91]. In general, WCET analysis has to deal with two problems[‡]:

1. Determine the timing behavior of if-then-else statements.
2. Determine the number of loop iterations.

Our simple approach handles case (item 1) by defining that (roughly speaking) the WCET of an if-then-else statement is equal to the maximum of the WCET of the then- and the else-branch. In case (item 2) we assume that there is an *oracle* which when given a certain loop statement, returns the (maximum) number of iterations of that loop. Usually the programmer has to state in the program code how often a certain loop iterates, i.e., it is assumed that the programmer has knowledge which cannot be derived from the source code alone. In this case the programmer takes the part of the oracle.

We define a semilattice based on set T, the set of *time values*. Time can be measured either by natural numbers (e.g. number of machine cycles) or by real numbers. Thus T is either equal to $\mathbb{N}_0$ or to $\mathbb{R}_0^+$.

[‡] We ignore recursive procedures here. Solutions to the problem of WCET analysis of recursive real-time procedures can be found in [BL95, BL96, Bli00].

We associate with each basic block B a number $\tau_B \in T$ which accounts for the time used by the execution of B. We assume that each τ_B is invariable, i.e., the timing behavior of a basic block does not change if it is executed several times. This excludes effects of caching or pipelining from our model. Some research on this subject has been conducted (cf. [HBW94, HWH95, LL94, AMWH94]) but these issues are out of the scope of this paper. In [BFS00] cache hit analysis is performed with help of symbolic evaluation.

The data-flow framework defined below will allow for estimating the overall timing behavior of a procedure by employing solution algorithms well-known for "classic" data-flow frameworks.

Definition 3.1.

$$L = \left\langle \mathbb{N}_0(\mathbb{R}_0^+), 0, \wedge \right\rangle$$

where $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$ is the set of non-negative integers, $\mathbb{R}_0^+ = \{x \geq 0 \mid x \in \mathbb{R}\}$ is the set of non-negative real numbers, and where for all $a, b \in \mathbb{N}_0(\mathbb{R}_0^+)$

$$a \wedge b := \max(a, b).$$

We call this operator the *max meet operator*.

The relation imposed by this meet operator is $\geq$.

Remark 3.1. Note that the lattice defined in Definition 3.1 is infinite, i.e., there is a $\underline{0}$ but no $\underline{1}$-element.

Properties 3.1.

1. L does not have d.c.c. because L is infinite.
2. L is closed under finite meets. L is not closed under infinite meets, e.g.

$$\bigwedge_{i=1}^{\infty} i \to \infty \notin L.$$

3. L is continuous for finite meets.

Definition 3.2. The function space F is defined by

$$F = \{f_c(x) = x + c\}, \quad \text{where } x, c \in \mathbb{N}_0(\mathbb{R}_0^+).$$

Properties 3.2.

1. F is monotone and distributive.

Proof. For all $f_c \in F$ and for all $x, y \in L$

$$x \geq y \Rightarrow f_c(x) = x + c \geq y + c = f_c(y).$$

Thus F is monotone.

For all $f_c \in F$ and for all $x, y \in L$

$$f_c(x \wedge y) = \max(x, y) + c = \max(x + c, y + c) = \max(f_c(x), f_c(y)) = f_c(x) \wedge f_c(y).$$

Thus F is distributive. □

2. F is continuous for finite meets.
3. F is not bounded.

Proof.

$$f_c^{[k]}(x) = \operatorname*{Max}_{i=0}^{k-1}\left(f_c^i(x)\right) = \operatorname*{Max}_{i=0}^{k-1}(x + i \cdot c) = x + \operatorname*{Max}_{i=0}^{k-1}(i \cdot c) = x + (k-1) \cdot c.$$

□

$$\begin{array}{lcl}
X_{\text{entry}} & = & \tau_{\text{entry}} \\
X_{\text{exit}} & = & \max(X_{\text{entry}}, X_2) + \tau_{\text{exit}} \\
X_1 & = & X_{\text{entry}} + \tau_1 \\
X_2 & = & \max(X_1, X_5) + \tau_2 \\
X_3 & = & X_2 + \tau_3 \\
X_4 & = & X_3 + \tau_4 \\
X_5 & = & \max(X_3, X_4) + \tau_5
\end{array}$$

Table 3. Set of Equations for Power Example

4. There does not exist a pseudo-transitive closure for F.

   Proof. A function $f^{@}$ does not exist for all $x \in L$, $i \geq 0$ such that

$$f_c^{@}(x) \geq f_c^{i}(x) = x + i \cdot c.$$

□

Definition 3.3. The mapping M assigns to an edge (B, B') the function $f_{\tau_B}(x) = x + \tau_B$.

Remark 3.2. Since $M(e)$ does not depend on the target, but only on the source of e, M could be viewed as defined on V alone.

Definition 3.4. Definition 3.3 results in the following set of WCET equations

$$\text{WCET}(B) = \underset{B' \in \text{Preds}(B)}{\text{Max}} \text{WCET}(B') + \tau_B$$
$$\text{WCET}(\rho) = \tau_\rho,$$

where $\text{Preds}(B)$ denotes the set of predecessors of B and $\tau_B \in T$ is the time used to execute basic block B.

Example 1. The set of equations for our example is shown in Table 3. For sake of simplicity we write X_i instead of $\text{WCET}(B_i)$.

From the properties derived above the following observation can be proved easily.

Observation 1. *If the control flow graph $G = (V, E, \rho)$ contains loops, the WCET data-flow framework defined in Definition 3.4 cannot be solved by* iteration algorithms*.*

Proof. Since L does not have d.c.c. and F is not bounded, each loop statement implies that the WCET-values increase permanently.

Hence the iteration does not converge.

As an example use the set of equations in Table 3 with arbitrary starting values. Insert these values on the right side of the equations, then perform this procedure with the new values, and so on. □

Remark 3.3. Note that iteration algorithms do not make use of the oracle. In contrast, as will be shown below, elimination methods can exploit this information quite well.

In the following we will show how the data-flow framework for WCET analysis can be solved by *elimination algorithms*. Our arguments are based on [Pau88].

The starting point of [Pau88] are recursive definitions or equations. These equations are solved by first building a dependency graph. The dependency graph contains an edge from x_1 to x_2 if the variable x_2 appears on the right side of the equation for x_1.

If the dependency graph is acyclic, the set of equations can be solved by simple insertions, thereby eliminating one variable after the other. If it contains cycles, insertions alone are not enough to obtain a solution. However, if a rule is available for replacing such an equation with one in which the left variable does not appear on the right, with a guarantee that any solution to this new equation set will satisfy the original, then it becomes possible to move the elimination process forward. Such a rule is called *loop-breaking rule*.

More formally, assume a set of equations E of the form

$$\{E_i : x_i = W_i(x_{i_1}, \dots, x_{i_{n_i}}) \mid i = 1, \dots, n\}$$

where E_i is the label for the ith equation. For each legitimate selection of values in their respective domains assigned to the right side variables $\{x_{i_1}, \dots, x_{i_{n_i}}\}$, $W_i(x_{i_1}, \dots, x_{i_{n_i}})$ is constructible and has a single value. The set of values that $W_i(x_{i_1}, \dots, x_{i_{n_i}})$ takes, as the right side variables assume all their legitimate values, has a partial ordering, inducing the relation $\leq$.

A *substitution transformation* of E, $s(E, i, j)$, for $1 \leq i, j \leq n$, yields the result of substituting the right side of E_i for an occurrence of x_i on the right side of equation E_j, $i \neq j$, and simplifying the resultant right side of E_j according to identities relating different expressions of W_i. $s(E, i, j)$ differs from E at most in having different E_j equations. $s(E, i, j) = E$ if x_i does not occur non-trivially in W_j.

An equation E_i is said to have a *loop-breaking rule* if there is an equation

$$e_i : x_i = w_i(x_{i_1}, \dots, x_{i_{n_i}}) \qquad (= b(E, i))$$

for x_i in which

LB-1: x_i does not occur (non-trivially) on the right of e_i.
LB-2: No variable that was not originally on the right is introduced.
LB-3: Every solution to e_i is a solution to E_i.
LB-4: For every solution S to E_i, there is a solution s of e_i so that $s \leq S$.

A set of equations E is said to have a loop-breaking rule if for each equation in E and for any equation resulting from E by a sequence of substitutional and loop-breaking transformations, there is a loop-breaking rule.

Properties 3.3.

1. Each solution of $s(E, i, j)$ is a solution of E and vice versa.
2. Each solution s of $b(E, i)$ is a solution to E_i, and for every solution S to E, there is a solution s of $b(E, i)$ so that $s \leq S$.
3. If a sequence of applications of transformations to a set of equations E produces the set E', then the solution to E' is also a solution to E, and if $\text{SOL} = \{x_i = S_i \mid i = 1, \dots, n\}$ is a solution to E, then there is a solution $\text{sol} = \{x_i = s_i \mid i = 1, \dots, n\}$ of E' with $\{s_i \leq S_i \mid i = 1, \dots, n\}$ (we say $\text{sol} \leq \text{SOL}$).

Proof. Properties 3.3.1–3 are proved in [Pau88]. □

A Gaussian-Elimination-Type algorithm can be used to solve sets of equations. For data-flow analysis the special structure of flow graphs can be exploited to construct algorithms with improved time complexity. For example see [AC76, HU77, Tar81a, GW76, Sre95, SGL98] and [RP86] for an overview of the first four algorithms. For our purposes we use the algorithm of [Sre95, SGL98].

Returning to our equations of Definition 3.4, we have to determine how to insert one equation into another, how to simplify our equations, and how to set up a loop-breaking rule.

We define the following *normal form* for our WCET equations:

Definition 3.5. A WCET equation is in *normal form* if it has the form

$$\mathrm{WCET}(B_i) = \operatorname*{Max}_{j \in J \subseteq \{1,\dots,n\}} \left(\mathrm{WCET}(B_j) + \tau_{n_j}\right)$$

where τ_{n_j} are expressions not involving WCET(.).

It is easy to see that an equation can always be brought to normal form.

Our loop-breaking rule is defined as follows:

Definition 3.6. Assume we have the following equation

$$E_i : \mathrm{WCET}(B_i) = \max\left(\mathrm{WCET}(B_i) + \tau_{n_i}, \operatorname*{Max}_{j \in J \subseteq \{1,\dots,n\}, j \neq i} \left(\mathrm{WCET}(B_j) + \tau_{n_j}\right)\right).$$

Then we replace it with

$$e_i = b(E, i) : \mathrm{WCET}(B_i) = \mathrm{ORACLE}(\mathrm{iter}) \cdot \tau_{n_i} + \operatorname*{Max}_{j \in J \subseteq \{1,\dots,n\}, j \neq i} \left(\mathrm{WCET}(B_j) + \tau_{n_j}\right),$$

where ORACLE(iter) denotes the number of iterations of the loop. Finally we bring equation e_i to normal form.

For WCET analysis conditions LB-3 and LB-4 cannot be interpreted in an appropriate manner. We therefore replace them with the following conditions:

WCET-LB-3: For equation E_i

$$E_i : x_i = W_i(x_{i_1}, \dots, x_i, \dots, x_{i_{n_i}})$$

define the recurrence relation R_i:

$$x_i(0) = W_i^{(0)}(x_{i_1}, \dots, x_{i_{n_i}}),$$
$$x_i(k+1) = W_i(x_{i_1}, \dots, x_i(k), \dots, x_{i_{n_i}})$$

where $W_i^{(0)}(\dots)$ is a function not depending on x_i which can somehow be derived from W_i.

Next consider o = ORACLE(iter) not a constant value returned by the oracle but a variable $o \in \mathbb{N}_0$. Then every solution $s_i(o)$ to equation e_i has to be a solution to R_i, i.e.,

$$s_i(0) = W_i^{(0)}(x_{i_1}, \dots, x_{i_{n_i}}),$$
$$s_i(o+1) = W_i(x_{i_1}, \dots, s_i(o), \dots, x_{i_{n_i}}).$$

WCET-LB-4: For every solution S to E_i, there is a solution s of e_i so that $s \geq S$.

Note that WCET-LB-4 differs from LB-4 only by replacing "$\leq$" with "$\geq$" and that Properties 3.3.1–3 can be proved with "$\leq$", LB-3 and LB-4 replaced with "$\geq$", WCET-LB-3 and WCET-LB-4, respectively.

It remains to show that our loop-breaking rule fulfills the four conditions LB-1, LB-2, WCET-LB-3, and WCET-LB-4. We obtain:

LB-1: WCET(B_i) does not occur on the right side of e_i.

LB-2: No additional variable is introduced on the right.

WCET-LB-3: Setting

$$W_i^{(0)}(\dots) = \bigwedge_{j \in J \subseteq \{1,\dots,n\}, j \neq i} \left(\mathrm{WCET}(B_j) + \tau_{n_j}\right),$$

it is easy to see that this condition is valid too.

In fact, every solution $s_i(o)$ with $o \geq$ ORACLE(iter) will be a solution to E_i.

$$\begin{array}{lcl}
X_{\text{entry}} & = & \tau_{\text{entry}} \\
X_{\text{exit}} & = & \max(X_{\text{entry}} + \tau_{\text{exit}}, X_2 + \tau_{\text{exit}}) \\
X_1 & = & X_{\text{entry}} + \tau_1 \\
X_2 & = & \max(X_1 + \tau_2, X_5 + \tau_2) \\
X_3 & = & X_2 + \tau_3 \\
X_4 & = & X_3 + \tau_4 \\
X_5 & = & \max(X_3 + \tau_5, X_4 + \tau_5)
\end{array}$$

TABLE 4. Set of Equations for Power Example in Normal Form

WCET-LB-4: This condition is also easy to verify.

Example 1. Returning to our example, the equations brought to normal form are shown in Table 4. Now, we solve this set of equations by applying the algorithm described in [Sre95, SGL98]. We write "$a \to b$" for indicating that equation E_a is inserted into E_b and we write "$c \not\circlearrowleft$" for loop-breaking equation E_c. Furthermore we use the abbreviation $\tau_{d,e}$ for $\tau_d + \tau_e$, which we generalize for arbitrary sets of subscripts.

We solve our set of equations assuming that $n = 11$, i.e., we compute an estimate for WCET of procedure **power** called with the arbitrary parameter x and $n = 11$.

$4 \to 5$**:**

$$X_5 = \max(X_3 + \tau_5, X_3 + \tau_{4,5}) = X_3 + \tau_{4,5}$$

by definition of the max meet operator.

$5 \to 2$**:**

$$X_2 = \max(X_1 + \tau_2, X_3 + \tau_{2,4,5})$$

$3 \to 2$**:**

$$X_2 = \max(X_2 + \tau_{2,3,4,5}, X_1 + \tau_2)$$

$2 \not\circlearrowleft$**:**

$$X_2 = \text{ORACLE(iter)} \cdot \tau_{2,3,4,5} + X_1 + \tau_2 = X_1 + \tau_2 + 4 \cdot \tau_{2,3,4,5}$$

$2 \to$ **exit:**

$$X_{\text{exit}} = \max(X_{\text{entry}} + \tau_{\text{exit}}, X_1 + \tau_{\text{exit},2} + 4 \cdot \tau_{2,3,4,5})$$

$1 \to$ **exit:**

$$X_{\text{exit}} = (X_{\text{entry}} + \tau_{\text{exit},1,2} + 4 \cdot \tau_{2,3,4,5})$$

Note that the contribution of the edge (entry $\to$ exit) disappears because of the max meet operator. If this would not have happened, we had to "subtract" it from the result below.

entry $\to$ exit:

$$X_{\text{exit}} = \tau_{\text{exit,entry},1,2} + 4 \cdot \tau_{2,3,4,5}$$

As can be easily verified, this equals the WCET estimate w.r.t. the max meet operator. Unfortunately the estimate ignores that the edge ($3 \to 5$) is followed one time and the path ($3 \to 4 \to 5$) is followed only three times.

As can be seen from the discussions and from the example above, elimination algorithms are well-suited for solving data-flow equations which describe the WCET behavior of procedures. However, one drawback of the max meet operator has been pointed out at the end of the example, namely that we do not get an exact WCET estimate. We will show in the following section how to obtain such an exact WCET estimate by employing a fully oracle-based approach.

$$\begin{array}{lcl}
P_{\text{entry}}(z) & = & z^{\tau_{\text{entry}}} \\
P_{\text{exit}}(z) & = & z^{\tau_{\text{exit}}}\left(0 \cdot P_{\text{entry}}(z) + \frac{1}{5} \cdot P_2(z)\right) \\
P_1(z) & = & z^{\tau_1} P_{\text{entry}}(z) \\
P_2(z) & = & z^{\tau_2}\left(P_1(z) + P_5(z)\right) \\
P_3(z) & = & z^{\tau_3} \frac{4}{5} P_2(z) \\
P_4(z) & = & z^{\tau_4} \frac{3}{4} P_3(z) \\
P_5(z) & = & z^{\tau_5}\left(\frac{1}{4} P_3(z) + P_4(z)\right)
\end{array}$$

TABLE 5. Set of Equations for Power Example

4. A Fully Oracle-Based Data-Flow Framework

In contrast to Section 3 we define an oracle-based meet operator $\wedge$. For each edge (B', B) the oracle returns a real value $0 \leq \text{ORACLE}(B', B) \leq 1$ such that

$$\sum_{B' \in \text{Succs}(B)} \text{ORACLE}(B, B') = 1.$$

$\text{ORACLE}(B, B')$ is a measure of how often the edge (B, B') is taken by executing the underlying procedure compared to the edges through the other successors of B. We call $\text{ORACLE}(B, B')$ *execution frequency* (compare [Ram96] for a similar approach).

By allowing $\text{ORACLE}(B, B') = 0$ we are able to model dead paths.

In contrast to Section 3 we set up equations of generating functions $P_i(z)$ in the following way.

Definition 4.1.

$$P_i(z) = z^{\tau_i} \sum_{j \in \text{Preds}(i)} \text{ORACLE}(j, i) P_j(z),$$
$$P_{\text{entry}}(z) = z^{\tau_{\text{entry}}}.$$

Again the underlying semilattice is not limited, which implies that the equations of Definition 4.1 cannot be solved by iteration algorithms.

Since the resulting equations are all linear equations, the loop breaking rule can be defined by simply solving the equation in a straight-forward way. Thus the solutions are *rational functions*.

Employing well-known facts of generating functions (cf. e.g. [GKP89]), the worst-case timing behavior of the underlying program is given by

$$\left.\frac{d}{dz} P_{\text{exit}}(z)\right|_{z=1}$$

which means that we differentiate P_{exit} w.r.t. z and then set $z = 1$.

Remark 4.1. Note that $P_B(1)$ equals the number of how often basic block B is executed.

Example 1. The set of equations of our power example ($n = 11$) take now the form shown in Table 5. Note that we have modeled the edge (entry $\rightarrow$ exit) as being a dead path by assigning a path execution frequency of zero.

Solving this set of equations again using the algorithm described in [Sre95, SGL98], we proceed as follows:

$4 \rightarrow 5$**:**

$$P_5(z) = z^{\tau_5}\left(\frac{1}{4} + z^{\tau_4}\frac{3}{4}\right) P_3(z)$$

$5 \to 2$:

$$P_2(z) = z^{\tau_2}\left(P_1(z) + z^{\tau_5}\left(\frac{1}{4} + z^{\tau_4}\frac{3}{4}\right)P_3(z)\right)$$

$3 \to 2$:

$$P_2(z) = z^{\tau_2}P_1(z) + z^{\tau_{2,3,5}}\left(\frac{1}{5} + z^{\tau_4}\frac{3}{5}\right)P_2(z)$$

$2\ \not\circlearrowleft$:

$$P_2(z) = \frac{z^{\tau_2}P_1(z)}{1 - z^{\tau_{2,3,5}}\left(\frac{1}{5} + z^{\tau_4}\frac{3}{5}\right)}$$

$2 \to$ exit:

$$P_{\text{exit}}(z) = \frac{1}{5}\cdot\frac{z^{\tau_{2,\text{exit}}}P_1(z)}{1 - z^{\tau_{2,3,5}}\left(\frac{1}{5} + z^{\tau_4}\frac{3}{5}\right)}$$

$1 \to$ exit:

$$P_{\text{exit}}(z) = \frac{1}{5}\cdot\frac{z^{\tau_{1,2,\text{exit}}}P_{\text{entry}}(z)}{1 - z^{\tau_{2,3,5}}\left(\frac{1}{5} + z^{\tau_4}\frac{3}{5}\right)}$$

entry $\to$ exit:

$$P_{\text{exit}}(z) = \frac{1}{5}\cdot\frac{z^{\tau_{\text{entry},1,2,\text{exit}}}}{1 - z^{\tau_{2,3,5}}\left(\frac{1}{5} + z^{\tau_4}\frac{3}{5}\right)}$$

Note that for example $P_2(1) = 5$, $P_3(1) = 4$, and $P_4(1) = 3$.

It remains to calculate

$$\frac{d}{dz}P_{\text{exit}}(z)\bigg|_{z=1} = \tau_{\text{entry}} + \tau_1 + 5\tau_2 + 4\tau_3 + 3\tau_4 + 4\tau_5 + \tau_{\text{exit}}.$$

This is the *exact timing behavior* of procedure **power** for $n = 11$.

In fact we have the following theorem.

Theorem 4.1. *Solving a set of WCET equations based on the frequency meet operator, we always obtain the exact timing behavior of the underlying procedure and not a less accurate estimate of the WCET.*

Proof. The proof is obvious from the definitions and from the discussion above, provided that the oracle always produces correct values, i.e., ORACLE(B', B) gives the exact path execution frequencies. $\square$

It remains to mention some results on the performance of elimination algorithms.

Theorem 4.2. *One of the algorithms presented in* [Sre95, SGL98] *solves data-flow equations in $O(\log|N| \cdot |E|)$ insertions and loop-breaking operations, where $|N|$ denotes the number of nodes in the CFG and $|E|$ is the number of edges of the CFG. The CFG is supposed to be reducible.*

The method described in [Tar81a] *takes $O(|E| \cdot \alpha(|E|, |N|))$ time, where α denotes the inverse Ackermann's function. Thus this algorithm behaves almost linear in time on reducible CFGs.*

Remark 4.2. If the oracle gives its answer in $O(1)$ time, we have at hand algorithms allowing efficient WCET analysis of (real-time) programs.

5. Getting Rid of the Oracle – A Symbolic Evaluation Approach

In this section we show how we can replace the oracle with *symbolic evaluation* and by solving *conditional recurrence relations.*

Before we give a formal foundation of our symbolic evaluation approach, we study our power example in detail to justify our theoretical treatment in the rest of the section.

Example 1. In contrast to Sections 3 and 4 we will not assume that $n = 11$, but we will derive *symbolic formulas* for the quantities of interest.

First we show how the number of loop iterations of procedure **power** can be determined. Now, the condition at the loop header ($e > 0$) is solely responsible for loop completion. Taking a closer look to the loop body, we see that there is a simple recurrence relation associated to variable e. Since $e = n$ before the loop is entered the first time, we obtain for $k \geq 0$:

$$\begin{aligned} e(0) &= n \\ e(k+1) &= \left\lfloor \frac{e(k)}{2} \right\rfloor \end{aligned} \tag{1}$$

This forms a strictly decreasing sequence of numbers and the loop completes if $e(\omega) = 0$, where ω denotes the smallest index fulfilling $e(\omega) = 0$.

If we can determine ω from the recurrence relation (1), we have found a symbolic formula for the number of iterations of the loop, which clearly depends on the initial value $e(0) = n$. Fortunately we can find a closed form expression for the solution of (1). Using well-known facts of the floor-function (cf. [GKP89]), we obtain for $k \geq 0$

$$e(k) = \left\lfloor \frac{n}{2^k} \right\rfloor . \tag{2}$$

Now $e(k)$ becomes zero if the nominator in (2) gets greater than the denominator, i.e., if

$$2^k > n.$$

Taking logarithms we get

$$\omega = \lfloor \operatorname{ld} n \rfloor + 1.$$

This is a reasonable simple formula, which the oracle can use for producing the number of loop iterations.

The other interesting quantity for determining the WCET of procedure **power** is how often Node 4 of the corresponding control flow graph is executed. We are successful if we can give a symbolic formula for this frequency in terms of n. Again we can set up a recurrence relation for the frequency $f(k)$, ($k \geq 0$):

$$\begin{aligned} f(0) &= 0, \\ f(k+1) &= \begin{cases} 1 + f(k), & \text{if } e(k) \bmod 2 = 1, \\ f(k), & \text{otherwise.} \end{cases} \end{aligned} \tag{3}$$

We can give a closed formula for $f(k)$ if we accept the function $\nu(k)$, which gives the number of ones in the binary representation of k, to be a "primitive function" that can be used in closed-form expressions. $\nu(k)$ can be defined by

$$\begin{aligned} \nu(0) &= 0, \\ \nu(2k) &= \nu(k), \\ \nu(2k+1) &= 1 + \nu(k). \end{aligned}$$

Then recurrence (3) is solved by

$$f(k) = \nu(k). \tag{4}$$

If $\nu(.)$ is not considered a "primitive function", the oracle can replace $\nu(k)$ with an upper bound of $\nu(k)$, thereby obtaining an upper bound for the WCET, and not a formula for the exact timing behavior. The proposed upper bound is

$$\nu(k) \leq \lfloor \operatorname{ld} k \rfloor + 1.$$

Summing up we either derive the exact formula

$$\text{WCET} = \tau_{\text{exit}} + \tau_{\text{entry}} + \tau_1 + \tau_2 + (\lfloor \operatorname{ld} n \rfloor + 1) \cdot (\tau_2 + \tau_3 + \tau_5) + \nu(n) \cdot \tau_4$$

syntax	semantics
null	$[\mathcal{S}_1, p_1] \cup \cdots \cup [\mathcal{S}_k, p_k]$
$s_1;s_2$	$\text{val}(s_2, \text{val}(s_1, [\mathcal{S}_1, p_1] \cup \cdots \cup [\mathcal{S}_k, p_k]))$
$v := E$	$[\text{vset}(\mathcal{S}_1, v, \text{val}(E, \mathcal{S}_1)), p_1] \cup \cdots \cup$ $[\text{vset}(\mathcal{S}_k, v, \text{val}(E, \mathcal{S}_k)), p_k]$

Table 6. Evaluating Simple Statements

or the upper bound

$$\text{WCET} \le \tau_{\text{exit}} + \tau_{\text{entry}} + \tau_1 + \tau_2 + (\lfloor \text{ld}\, n \rfloor + 1) \cdot (\tau_2 + \tau_3 + \tau_4 + \tau_5).$$

5.1. **Symbolic Evaluation.** *Symbolic evaluation* is a form of static program analysis in which symbolic expressions are used to denote the values of program variables and computations (cf. e.g. [CHT79, Plo80, CR81, Sch96, BFS00, BB98, BBS99, BBS00, FS97]). In addition a path condition describes the impact of the program's control flow onto the values of variables and the condition under which control flow reaches a given program point.

5.1.1. *Program State and Context.* The *state* $\mathcal{S}$ of a program is a set of pairs $\{(v_1, e_1), \ldots, (v_m, e_m)\}$ where v_i is a program variable and e_i is a symbolic expression describing the value of v_i for $1 \le i \le m$. For each variable v_i there exists exactly one pair (v_i, e_i) in $\mathcal{S}$.

A program consists of a sequence of statements that may change $\mathcal{S}$.

A *path condition* specifies a condition that is valid at a certain program point. If conditional statements are present, there may be several different valid program states at the same program point. A different path condition is associated with each of them.

Pairs of states $\mathcal{S}$ and path conditions $\mathcal{C}$ specify a *program context* which is defined by

$$\bigcup_{i=1}^{k} [\mathcal{S}_i, \mathcal{C}_i]$$

where k denotes the number of different program states valid at a certain program point. A program context completely describes the variable bindings at a specific program point together with the associated path conditions.

The function $\text{vget}(\mathcal{S}, v)$ is used to extract the value of variable v from program state $\mathcal{S}$, $\text{vset}(\mathcal{S}, v, n)$ returns the state that is implied by setting the value of v in $\mathcal{S}$ to n.

5.1.2. *Expressions and Simple Statements.* Expressions are evaluated symbolically in such a straightforward manner that we do not elaborate on details. The interested reader is referred to [CHT79, FS97] for a more complete treatment of this issue for certain programming languages.

Examples for simple statements are given in Table 6. We use $\text{val}(\ldots)$ for denoting symbolic evaluation of program contexts. Of course, details of evaluating simple statements may differ from programming language to programming language, thus, again, we do not go too deeply into details.

5.1.3. *Conditional Statements.* In case of an if-statement both branches are evaluated according to

$$\begin{aligned}\mathrm{val}(\texttt{if}\ c\ \texttt{then}\ s_1\ \texttt{else}\ s_2, [\mathcal{S}_1, p_1] \cup \cdots \cup [\mathcal{S}_k, p_k]) \mapsto \\ \mathrm{val}(s_1, [\mathcal{S}_1, p_1 \wedge \mathrm{val}(c, \mathcal{S}_1)] \cup \cdots \cup \\ [\mathcal{S}_k, p_k \wedge \mathrm{val}(c, \mathcal{S}_k)]) \cup \\ \mathrm{val}(s_2, [\mathcal{S}_1, p_1 \wedge \neg\, \mathrm{val}(c, \mathcal{S}_1)] \cup \cdots \cup \\ [\mathcal{S}_k, p_k \wedge \neg\, \mathrm{val}(c, \mathcal{S}_k)]).\end{aligned}$$

Case-statements can be treated in a similar way.

5.1.4. *Loop Statements.* For a while-statement there are two cases:

1. the loop terminates and
2. the loop does not terminate.

In case (item 1) the path condition and the negated loop condition build the new path condition. In case (item 2) the loop is evaluated recursively and the path condition is extended by the (non-negated) loop condition.

$$\begin{aligned}\mathrm{val}(\texttt{while}\ c\ \texttt{do}\ s, [\mathcal{S}_1, p_1] \cup \cdots \cup [\mathcal{S}_k, p_k]) \mapsto \\ \mathrm{val}(\texttt{while}\ c\ \texttt{do}\ s, \\ \mathrm{val}(s, [\mathcal{S}_1, p_1 \wedge \mathrm{val}(c, \mathcal{S}_1)] \cup \cdots \cup \\ [\mathcal{S}_k, p_k \wedge \mathrm{val}(c, \mathcal{S}_k)]) \cup \\ [\mathcal{S}_1, p_1 \wedge \neg\, \mathrm{val}(c, \mathcal{S}_1)] \cup \cdots \cup \\ [\mathcal{S}_k, p_k \wedge \neg\, \mathrm{val}(c, \mathcal{S}_k)])\end{aligned}$$

Clearly in only very rare cases we will obtain a symbolically decidable loop. For all the other loops we have to use alternate approaches. We will elaborate on this issue later (cf. Definition 5.4).

Other loop-statements can be treated similarly.

5.2. **A Data-Flow Framework for Symbolic Evaluation.** In order to set up a data-flow framework for symbolic evaluation of programs, we first note that a basic block contains simple statements and expressions only. Control flow affecting statements (ifs, loops, ...) are mirrored by the structure of the control flow graph. The conditions used for affecting the control flow, however, are evaluated at the end of the basic block which has more than one successor. We assume in the following that an edge (of the CFG) $e = (B', B)$ has assigned a condition $\mathrm{Cond}(B', B)$ which must evaluate to true for the control flow to follow this edge. In case of the then-branch of an if-statement Cond is the condition of the if-statement, in case of the else-branch it is the negated condition of the if-statement, in case of a while-loop Cond is the condition at the loop header for the edge going to the loop body, and for the edge not going to the loop body it equals the exit condition. For-loops can be modeled similarly.

In case of general loops with exit-statements Cond is the condition at the loop header for the edge going to the loop body, and for the edge not going to the loop body (if any) it equals the negated condition at the loop header. For each edge leaving the loop its Cond is the corresponding exit condition.

In the following we assume that each basic block does not have more than two successors[§].

Definition 5.1. We start by defining a meet semilattice for program contexts as follows

$$L = \langle A, \underline{0}, \underline{1}, \cup \rangle\,,$$

[§] This assumption is no real restriction since it can be generalized to any number of successors in a straightforward manner.

where A denotes the power set of program contexts, $\underline{0}$ is the set of all program contexts, $\underline{1}$ is the empty set, and $\cup$ is the *symeval meet operator*.

Definition 5.2. We define the following set of equations for the symbolic evaluation framework:

$$\mathrm{SymEval}(\rho) = [\mathcal{S}_0, \mathcal{C}_0],$$

where $\mathcal{S}_0$ denotes the initial state containing all variables which are assigned their initial values, and $\mathcal{C}_0$ is true,

$$\mathrm{SymEval}(B) = \bigcup_{B' \in \mathrm{Preds}(B)} \mathrm{PrpgtCond}(B', B, \mathrm{SymEval}(B')) \mid \mathrm{LocalEval}(B),$$

where $\mathrm{LocalEval}(B) = \{(v_{i_1}, e_{i_1}), \dots, (v_{i_m}, e_{i_m})\}$ denotes the symbolic evaluation local to basic block B. The variables that get a new value assigned in the basic block are denoted by $v_{i_1}, \dots, v_{i_m}$. The new symbolic values are given by $e_{i_1}, \dots, e_{i_m}$. It is important to note that the variables contained in the symbolic expressions are either global variables, parameters or variables getting assigned a symbolic value before this basic block is executed. The *propagated conditions* are defined by

$$\mathrm{PrpgtCond}(B', B, \mathrm{PC}) = \begin{cases} \mathrm{Cond}(B', B) \odot \mathrm{PC}, & \text{if } B' \text{ has two successors,} \\ \mathrm{PC}, & \text{otherwise.} \end{cases}$$

Denoting by PC a program context, the operation $\odot$ is defined as follows:

$$\begin{aligned} \mathrm{Cond}(B', B) \odot \mathrm{PC} &= \mathrm{Cond}(B', B) \odot [\mathcal{S}_1, p_1] \cup \dots \cup [\mathcal{S}_k, p_k] \\ &= [\mathcal{S}_1, \mathrm{Cond}(B', B) \wedge p_1] \cup \dots \cup [\mathcal{S}_k, \mathrm{Cond}(B', B) \wedge p_k]. \end{aligned}$$

Note that the definition of $\mathrm{PrpgtCond}(\dots)$ prevents the symbolic evaluation framework from being bounded. Thus it cannot be solved by iteration algorithms. By defining a suitable loop-breaking rule, however, we can still employ elimination algorithms.

Before we give our loop-breaking rule, we define a normal form for SymEval equations.

Definition 5.3. A SymEval equation E_i is in *normal form* if it has the form

$$E_i : \mathrm{SymEval}(B_i) = \bigcup_{1 \le j \le r, 1 \le i_j \le n} \big(C_j \odot \mathrm{SymEval}(B_{i_j})\big) \big| \{(v_{j_1}, e_{j_1}), \dots, (v_{j_m}, e_{j_m})\},$$

where $1 \le j_k \le m$.

Remark 5.1. Note that it is possible to bring any SymEval equation to normal form because of the following properties of the involved operators (the C_i are conditions, PC_i are program contexts, and $\mathrm{LocalEval}_i$ denote local symbolic evaluations):

$$C_1 \odot (C_2 \odot \mathrm{PC}_1) = (C_1 \wedge C_2) \odot \mathrm{PC}_1$$

$$(C_1 \odot \mathrm{PC}_1 \mid \mathrm{LocalEval}_1) \mid \mathrm{LocalEval}_2 = C_1 \odot \mathrm{PC}_1 \mid (\mathrm{LocalEval}_1 \mid \mathrm{LocalEval}_2)$$

$$C_1 \odot (\mathrm{PC}_1 \cup \mathrm{PC}_2) = (C_1 \odot \mathrm{PC}_1) \cup (C_1 \odot \mathrm{PC}_2)$$

$$(\mathrm{PC}_1 \cup \mathrm{PC}_2) \mid \mathrm{LocalEval}_1 = (\mathrm{PC}_1 \mid \mathrm{LocalEval}_1) \cup (\mathrm{PC}_2 \mid \mathrm{LocalEval}_1)$$

Definition 5.4. The loop-breaking rule for SymEval equations is defined as follows: Assume we have the following equation (we use X_i for SymEval(B_i) as a shorthand)

$$E_i : X_i = \bigcup_{1\leq j\leq r, 1\leq j_i\leq n} (C_j \odot X_i) \mid \{(v_{j_1}, e_{j_1}), \ldots, (v_{j_s}, e_{j_s})\} \cup \bigcup_{1\leq k\leq t, 1\leq i_k\leq n, i_k\neq i, 1\leq k\leq n} (C_k \odot X_{i_k}) \mid \{(v_{k_1}, e_{k_1}), \ldots, (v_{k_u}, e_{k_u})\},$$

then we replace it with

$$e_i : X_i = \bigcup_{1\leq k\leq t, 1\leq i_k\leq n, i_k\neq i, 1\leq k\leq n} ((C_k \odot X_{i_k}) \mid \{(v_{k_1}, e_{k_1}), \ldots, (v_{k_u}, e_{k_u})\}) \mid \text{LoopExit},$$

where

$$\text{LoopExit} = \{(v_{j_1}, v_{j_1}(\perp, \omega_\ell)), \ldots, (v_{j_t}, v_{j_t}(\perp, \omega_\ell))\}$$

for all variables v_{j_i} being contained in

$$\bigcup_{1\leq j\leq r} \{(v_{j_1}, e_{j_1}), \ldots, (v_{j_s}, e_{j_s})\}.$$

The purpose of our loop-breaking rule is to replace a loop by a *set of recurrence relations*. Each induction variable (cf. [ASU86]) gives raise to an (indirect) recursion. Let v be such a variable, then $v(v_0, \omega_\ell)$ denotes the symbolic solution of the recursion, where v_0 is a suitable initial value and ω_ℓ denotes the number of iterations of loop ℓ[¶] ($v(v_0, 0) = v_0$). If no initial value is known or the initial value is irrelevant to the solution, we write $v(\perp, \omega_\ell)$.

It is easy to see that Definition 5.4 fulfills condition LB-1 to LB-4 in Section 3. Thus, as a byproduct, we have the following observation.

Observation 2. *Symbolic evaluation can be performed on reducible and irreducible control flow graphs.*

Proof. The proof follows immediately from the fact that for instance the algorithm for solving data-flow equations by elimination presented in [Sre95, SGL98] is capable of handling both reducible and irreducible flow graphs. $\square$

In order to perform symbolic evaluation according to the SymEval framework, we still have to clarify

- how to set up the recurrence relation during loop-breaking,
- how to handle $\{\ldots, (v, e_1), \ldots\} \mid \{\ldots, (v, e_2), \ldots\}$,
- $\{\ldots, (v, e), \ldots\} \mid \{\ldots, (w, e(v)), \ldots\}$,
- $\{\ldots, (v, e), \ldots\} \mid \{\ldots, (v, v(\perp, \omega)), \ldots\}$,
- $[\{\ldots, (v, e), \ldots\}, C(\ldots, v, \ldots)]$,
- $[\{\ldots, (v, v(v_0, \omega)), \ldots\}, C(\ldots, v, \ldots)]$,
- $[\{\ldots, (v, v(v_0, \omega)), \ldots\}, C(\ldots, v(\perp, \omega), \ldots)]$, and
- how to handle arrays and pointers.

Definition 5.5.

1. *Sequence* – If a situation like

$$\{\ldots, (v, e_1), \ldots\} \mid \{\ldots, (v, e_2), \ldots\},$$

is encountered during symbolic evaluation, we replace it with

$$\{\ldots, (v, e_2), \ldots\}.$$

The pair (v, e_1) is not contained in the new set.

[¶] Each loop gets assigned a unique number $\ell \in \mathbb{N}$.

2. *Expression Substitution* – If a situation like
$$\{\dots,(v_1,e_1),\dots\} \mid \{\dots,(v_2,e_2(v_1)),\dots\},$$
where $e(v)$ denotes an expression involving variable v, is encountered during symbolic evaluation, we replace it with
$$\{\dots,(v_1,e_1),\dots,(v_2,e_2(e_1)),\dots\}.$$
3. *Initial Iteration Value* – If a situation like
$$\{\dots,(v,e),\dots\} \mid \{\dots,(v,v(\perp,\omega)),\dots\}$$
is encountered during symbolic evaluation, we replace it with
$$\{\dots,(v,v(e,\omega)),\dots\}.$$
The pair (v,e) is not contained in the new set.

   For the situations discussed above it is important to apply the rules in the correct order, which is to elaborate the elements of the right set from left to right.
4. *Condition Substitution* – If a situation like
$$[\{\dots,(v,e),\dots\},C(\dots,v,\dots)]$$
is encountered during symbolic evaluation, we replace it with
$$[\{\dots,(v,e),\dots\},C(\dots,e,\dots)].$$
5. *Iteration Entry Condition* – If a situation like
$$[\{\dots,(v,v(v_0,\omega)),\dots\},C(\dots,v,\dots)]$$
is encountered during symbolic evaluation, we replace it with
$$[\{\dots,(v,v(v_0,\omega)),\dots\},C(\dots,v_0,\dots)].$$
6. *Nested Loop Entry Condition* – If a situation like
$$[\{\dots,(v,v(v_0,\omega)),\dots\},C(\dots,v(\perp,\omega),\dots)]$$
is encountered during symbolic evaluation, we replace it with
$$[\{\dots,(v,v(v_0,\omega)),\dots\},C(\dots,v(v_0,\omega),\dots)]$$

Setting up recurrence relations during loop-breaking is described in the following. If there are nested loops in the source code of interest, we start by setting up recurrence relations from the innermost loop and proceed to the outermost[‖].

Definition 5.6. Let v denote a variable, then we call $v(k)$ its *recursive counterpart*.

According to the notation in Definition 5.4 we set up a recurrence relation for all $1 \le j \le r$, $1 \le q \le s$ and for $k \ge 0$ by
$$v_{j_q}(k+1) = e_{j_q}(k) \qquad \text{if } C_j(k) \text{ evaluates to true,}$$
where $e_{j_q}(k)$ and $C_j(k)$ means that all variables contained in e_{j_q} and C_j are replaced with their recursive counterparts.

Note that we have not specified initial values for the recursion; these are supposed to be supplied by situations handled by Definition 5.5.

Since the expressive power of the system of conditional recurrence relations defined in Definition 5.6 is equal to that of recursive functions with the μ operator (cf. [Rog92]), solving such systems of recurrence relations is undecidable; in fact it is equivalent to the halting problem. However, for simple loops, such as for-loops or discrete loops (cf. [Bli94]), which generalize for-loops, the corresponding systems of recurrence relations can be solved.

[‖] This is guaranteed by the algorithm described in [Sre95, SGL98].

The undecidability of the above problem shows that we cannot get rid of the oracle completely, but certainly today's real-time programming languages and their designers place too heavy a burden on the oracle (the programmer) by letting it alone determine the number of loop iterations (cf. e.g. [PK89, HS91, Sha89, ITM90, GR91]).

The time complexity of symbolic evaluation of a reducible CFG is the same as that given by Theorem 4.2 in terms of insertions and loop-breaking operations. Solving conditional recurrence relations, however, may take considerably more time. In fact, since solving such recursions is undecidable, we cannot give time-bounds for this task. Hence we cannot set up bounds for the time used by the oracle to give its answers.

Finally arrays and pointers are an extremely difficult problem for symbolic evaluation. Although there has been some progress in research on array privatization (cf. e.g. [MAL93, Li92]) and the aliasing problem induced by dynamic data structures (cf. e.g. [Lan92, Ram94, LR92, Deu94]), we do not pursue these issues in this paper. Consequently in the rest of the paper arrays showing up in Example 3 are handled up to a certain degree of complexity only. To be more specific, recurrence relations on arrays are not solved explicitely and array subscripts are treated only if they are constant or of simple structure. Pointers are not treated at all because they are not contained in any example of this paper. How arrays can be treated in a symbolic evaluation framework is described and heavily used in [BFS00]. The same ideas can be used to model aliasing effects [BBS99] and pointers [SBF00].

5.3. **Symbolic Instrumentation.**

In this section we describe a method for determining symbolic path execution frequencies. We call the method *symbolic instrumentation.*

Definition 5.7. For each basic block B_i of a CFG we define a symbolic integer-valued variable b_i. The initial value of b_i is zero and by entering basic block B_i, the variable b_i is incremented by one.

We call such a CFG *symbolically instrumented* and b_i an *instrumentation variable* and its symbolic value after symbolic evaluation *basic block execution frequency.*

We have the following theorem.

Theorem 5.1. *By symbolic evaluation of a symbolically instrumented CFG we get symbolic formulas for the basic block execution frequencies (provided all recurrence relations can be solved).*

Proof. After symbolic evaluation of the instrumented CFG the symbolic formulas of the variables b_i describe how often the corresponding basic block B_i will be executed during runtime. □

Symbolic instrumentation can also handle finding safe upper bounds for conditional recurrence relations by exploiting the structure of the underlying CFG. Before we give the corresponding result, we need some definitions.

Definition 5.8. In a CFG, a node x *dominates* another node y iff all paths from the entry node to y always pass through x. We write $x \operatorname{dom} y$ in this case.

If $x \operatorname{dom} y$ and $x \neq y$, then x *strictly dominates* y. We write $x \operatorname{stdom} y$ to indicate that x strictly dominates y.

A node x is said to *immediately dominate* another node y, if $x \operatorname{stdom} y$ and there is no other node $z \neq x$ and $z \neq y$ such that $x \operatorname{stdom} z \operatorname{stdom} y$.

Theorem 5.2. *The solution of the instrumentation variable b_j of basic block B_j can be used as a safe upper bound for the solution of b_i, the instrumentation variable of basic block B_i, if B_j is the immediate dominator of B_i, B_i and B_j are contained*

$$\begin{array}{lcl}
X_{\text{entry}} & = & [\{(e,\perp),(h,\perp),(l,\perp),(y,\perp),(b_3,\perp),(b_4,\perp)\},\text{true}] \\
X_{\text{exit}} & = & (\neg(e>0)) \odot X_2 \\
X_1 & = & X_{\text{entry}} \mid \{(l,x),(e,n),(y,1)\} \\
X_2 & = & X_1 \cup X_5 \\
X_3 & = & ((e>0)\odot X_2) \mid \{(b_3,b_3+1),(h,e \bmod 2),(e,e/2)\} \\
X_4 & = & ((h=1)\odot X_3) \mid \{(b_4,b_4+1),(y,y\cdot l)\} \\
X_5 & = & (X_4 \mid \{(l,l\cdot l)\}) \cup (\neg(h=1)\odot X_3 \mid \{(l,l\cdot l)\})
\end{array}$$

TABLE 7. Set of SymEval Equations for Power Example

in the same loop, and B_j is not the loop header node of a for- or while-loop. If B_j is such a loop header node, $b_i = b_j - 1$. If B_j is in a loop, but B_i is not in the same loop, b_i can be bounded by the instrumentation variable of the immediate dominator of the loop header of the loop containing B_j.

Proof. The proof is straightforward by definition of immediate dominators (see Definition 5.8 or [ASU86]) and instrumentation variables. □

Remark 5.2. Note that the dominator tree, a tree whose edges reflect the immediate dominance relation, can be constructed in linear time (see [ALT96]).

If we have symbolic expressions for all instrumentation variables b_i, we can set up equations

$$b_i = \sum_{B_j \in \text{Succs}(B_i)} c_{ij}$$

$$b_i = \sum_{B_k \in \text{Preds}(B_i)} c_{ki} \tag{5}$$

where c_{ij} is a symbolic counter assigned to edge $e_{ij} = (B_i, B_j)$, $\text{Preds}(B_i)$ and $\text{Succs}(B_i)$ denote the set of predecessors and successors of B_i, respectively.

Note that each variable c_{ij} occurs exactly two times in the set of equations (5).

Furthermore note that if we make some variable explicit

$$c_{ij} = e_{ij}$$

and insert it for the other occurrence of c_{ij}, we obtain a new set of equations with the number of variables reduced by one, but again each variable occurs exactly two times.

Performing this reduction process iteratively until only one variable is left and doing a backward substitution thereafter, we see that the system of equations (5) can be solved in $O(|E|)$ steps. Thus we have proved the following theorem.

Theorem 5.3. *If symbolic expressions for all instrumentation variables of a CFG are available, symbolic expressions for all path execution frequencies can be determined in $O(|E|)$ time, where $|E|$ denotes the number of edges of the CFG.*

Remark 5.3. If only symbolic upper bounds for some b_i in equation (5) are known, we obtain a system of inequalities. However, it is easy to see that solving this system after replacing all "$\leq$" by "$=$", produces symbolic upper bounds for the path execution frequencies.

In the following we show how symbolic instrumentation works in practice by applying it to two examples, our power example and *Heapsort*.

Example 1. To keep the number of variables small, we will perform computations only for two instrumentation variables of our power example, namely for b_3 and

b_4. The set of SymEval equations is given in Table 7. The symbol $\perp$ is used to denote undefined values. Note that the edge (entry $\rightarrow$ exit) has been assigned Cond = false.

Solving this set of equations again using the algorithm described in [Sre95, SGL98], we proceed as follows:

$4 \rightarrow 5$**:**

$$\begin{aligned} X_5 = &((h=1) \odot X_3) \mid \{(b_4, b_4+1), (y, y \cdot l), (l, l \cdot l)\} \cup \\ &(\neg(h=1) \odot X_3) \mid \{(l, l \cdot l)\} \end{aligned}$$

$5 \rightarrow 2$**:**

$$\begin{aligned} X_2 = X_1 \cup &((h=1) \odot X_3) \mid \{(b_4, b_4+1), (y, y \cdot l), (l, l \cdot l)\} \cup \\ &(\neg(h=1) \odot X_3) \mid \{(l, l \cdot l)\} \end{aligned}$$

$3 \rightarrow 2$**:**

$$\begin{aligned} X_2 = X_1 \cup &((h=1) \wedge (e>0) \odot X_2) \mid \\ &\{(b_3, b_3+1), (h, e \bmod 2), (e, e/2), (b_4, b_4+1), (y, y \cdot l), (l, l \cdot l)\} \cup \\ &(\neg(h=1) \wedge (e>0) \odot X_2) \mid \\ &\{(b_3, b_3+1), (h, e \bmod 2), (e, e/2), (l, l \cdot l)\} \end{aligned}$$

$2 \not\circlearrowleft$**:**

$$\begin{aligned} X_2 = X_1 \mid &\{(b_3, b_3(\perp, \omega)), (h, h(\perp, \omega)), (e, e(\perp, \omega)), \\ &(b_4, b_4(\perp, \omega)), (y, y(\perp, \omega)), (l, l(\perp, \omega))\} \end{aligned}$$

$2 \rightarrow$ **exit:**

$$\begin{aligned} X_{\text{exit}} = &(\neg(e(\perp, \omega) > 0)) \odot X_1 \mid \\ &\{(b_3, b_3(\perp, \omega)), (h, h(\perp, \omega)), (e, e(\perp, \omega)), \\ &(b_4, b_4(\perp, \omega)), (y, y(\perp, \omega)), (l, l(\perp, \omega))\} \end{aligned}$$

$1 \rightarrow$ **exit:**

$$\begin{aligned} X_{\text{exit}} = &(\neg(e(n, \omega) > 0)) \odot X_{\text{entry}} \mid \\ &\{(b_3, b_3(\perp, \omega)), (h, h(\perp, \omega)), (e, e(n, \omega)), \\ &(b_4, b_4(\perp, \omega)), (y, y(1, \omega)), (l, l(x, \omega))\} \end{aligned}$$

Note that we have used rule item 3 of Definition 5.5.

entry $\rightarrow$ exit:

$$\begin{aligned} X_{\text{exit}} = \quad &[\{(e, e(n, \omega)), (h, h(\perp, \omega)), (l, l(x, \omega)) , \\ &(y, y(1, \omega)), (b_3, b_3(0, \omega)), (b_4, b_4(0, \omega))\}, (n > 0)] \end{aligned}$$

Note that we have used rules item 4 and item 6 of Definition 5.5.

The recurrence relations for the variable e has already been studied at the beginning of Section 5. According to Definition 5.6 we obtain the following conditional recurrence relations for b_3 and b_4:

$$b_3(k+1) = b_3(k) + 1 \quad \text{if } e(k) > 0$$

and

$$b_4(k+1) = b_4(k) + 1 \quad \text{if } (e(k) \bmod 2 = 1) \wedge (e(k) > 0).$$

Using the initial values we get

$$b_3(0, \omega) = b_3(0, \lfloor \operatorname{ld} n \rfloor + 1) = \lfloor \operatorname{ld} n \rfloor + 1$$

```
N: constant positive := ??;          -- number of elements to be sorted
subtype index is positive range 1 .. N;
type sort_array is array(index) of integer;

procedure heapsort(
    arr: in out sort_array) is

  N: index := arr'length;
  t: index;

  procedure siftdown(N,k:index) is
    j: index;
    v: integer;
  begin
    v := arr(k);
    discrete h := k in 1..N/2 new h := 2*h | 2*h+1 loop
      j := 2*h;
      if j<N and then arr(j)<arr(j+1) then
        j := j+1;
      end if;
      if v ≥ arr(j) then
        arr(h) := v;
        exit;
      end if;
      arr(h) := arr(j);
      arr(j) := v;
      h := j;
    end loop;
  end siftdown;

begin                                                        -- heapsort
  for k in reverse 1..N/2 loop
    siftdown(N,k);
  end loop;
  for M in reverse 2..N loop
    t := arr(1);
    arr(1) := arr(M);
    arr(M) := t;
    siftdown(M-1,1);
  end loop;
end heapsort;
```

FIGURE 3. Implementation of Heapsort in Ada

and

$$b_4(0,\omega) = b_4(0, \lfloor \operatorname{ld} n \rfloor + 1) = \nu(n) \leq \lfloor \operatorname{ld} n \rfloor + 1,$$

where the upper bound has been obtained by exploiting the fact that Node 3 is the immediate dominator of Node 4.

As a final remark note that we did not have to find solutions to all recurrence relations. For example variables l and y do not affect path execution frequencies. In a sense our approach requires "minimal" knowledge to determine path execution frequencies.

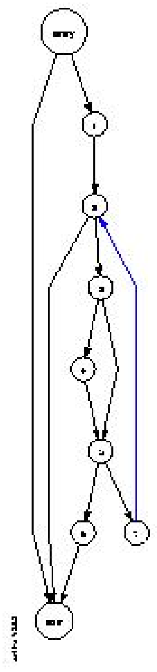

FIGURE 4. CFG of Procedure **siftdown**

$$
\begin{array}{lcl}
X_{\text{entry}} & = & [\{(arr, \underline{arr}), (N, \underline{N}), (k, \underline{k}), (h, \bot), (j, \bot), (v, \bot), (b_3, 0)\}, \text{true}] \\
X_{\text{exit}} & = & (\neg C_1) \odot X_2 \cup X_6 \\
X_1 & = & X_{\text{entry}} \mid \{(v, arr(k)), (h, k))\} \\
X_2 & = & X_1 \cup X_7 \\
X_3 & = & C_1 \odot X_2 \mid \{(b_3, b_3 + 1), (j, 2 \cdot h)\} \\
X_4 & = & C_2 \odot X_3 \mid \{(j, j + 1)\} \\
X_5 & = & X_4 \cup \neg C_2 \odot X_3 \\
X_6 & = & C_{3,\omega}(\bot, \bot) \odot X_5 \mid \{(arr(h), v)\} \\
X_7 & = & \neg C_3 \odot X_5 \mid \{(arr(h), (arr(j)), (arr(j), v), (h, j)\}
\end{array}
$$

TABLE 8. Set of SymEval Equations for Procedure **siftdown**

Example 3. *Heapsort* is a well-known and well-studied sorting algorithm (compare [Knu73b, Sed88, SS93]). We are now going to apply symbolic instrumentation to the implementation shown in Figure 3.

In this section we concentrate on procedure **siftdown**; the example will be completed in Section 5.4. The corresponding CFG is shown in Figure 4. A discrete loop (cf. [Bli94]) is used in **siftdown**. First, we ignore its effect by simply assuming that it is replaced with

```
h := k;
while h in 1 .. N/2 loop
```

where h is a variable of type *Index*.

The SymEval equations for **siftdown** are given in Table 8. We have used the following abbreviations:

$$\begin{aligned}
C_1 &= 1 \leq h \leq N/2,\\
C_{1,\omega}(h_0) &= 1 \leq h(h_0,\omega) \leq N/2,\\
C_2 &= (j < N)\overline{\wedge}(arr(j) < arr(j+1)),\\
C_3 &= v \geq arr(j),\\
C_{3,\omega}(v_0, j_0) &= v(v_0,\omega) \geq arr(j(j_0,\omega)),
\end{aligned}$$

where $\overline{\wedge}$ is semantically equivalent to Ada's **and then** operator, and we have used underlining for denoting the values of parameters and global variables. Note that we have to use $C_{3,\omega}(\perp,\perp)$ for the definition of equation X_6 since $5 \to 6$ is an edge leaving the loop via an exit-statement. Note also that we have restricted our interest to the instrumentation variable b_3 to keep the example small.

The solving procedure according to [Sre95, SGL98] is:

$7 \to 2$**:**

$$X_2 = X_1 \cup \neg C_3 \odot X_5 \mid \{(arr(h),(arr(j)),(arr(j),v),(h,j)\}$$

$6 \to$ **exit:**

$$\begin{aligned}
X_{\text{exit}} =& (\neg C_1) \odot X_2 \cup\\
& C_{3,\omega}(\perp,\perp) \odot X_5 \mid \{(arr(h),v)\}
\end{aligned}$$

$4 \to 5$**:**

$$X_5 = C_2 \odot X_3 \mid \{(j,j+1)\} \cup \neg C_2 \odot X_3$$

$5 \to 2$**:**

$$\begin{aligned}
X_2 =& X_1 \cup\\
& (\neg C_3 \wedge C_2) \odot X_3 \mid\\
&\quad \{(j,j+1),(arr(h),(arr(j+1)),(arr(j+1),v),(h,j+1)\} \cup\\
& (\neg C_3 \wedge \neg C_2) \odot X_3 \mid\\
&\quad \{(arr(h),(arr(j)),(arr(j),v),(h,j))\}
\end{aligned}$$

$5 \to$ **exit:**

$$\begin{aligned}
X_{\text{exit}} =& (\neg C_1) \odot X_2 \cup\\
& (C_{3,\omega}(\perp,\perp) \wedge C_2) \odot X_3 \mid \{(j,j+1),(arr(h),v)\} \cup\\
& (C_{3,\omega}(\perp,\perp) \wedge \neg C_2) \odot X_3 \mid \{(arr(h),v)\}
\end{aligned}$$

$3 \to 2$**:**

$$\begin{aligned}
X_2 = X_1 \cup & (\neg C_3 \wedge C_2 \wedge C_1) \odot X_2 \mid \{(b_3,b_3+1),(j,2\cdot h+1),\\
&\quad (arr(h),arr(2\cdot h+1)),(arr(2\cdot h+1),v),(h,2\cdot h+1)\} \cup\\
& (\neg C_3 \wedge \neg C_2 \wedge C_1) \odot X_2 \mid \{(b_3,b_3+1),(j,2\cdot h),\\
&\quad (arr(h),(arr(2\cdot h)),(arr(2\cdot h),v),(h,2\cdot h))\}
\end{aligned}$$

$3 \to$ **exit:**

$$\begin{aligned}
X_{\text{exit}} =& (\neg C_1) \odot X_2 \cup\\
& (C_{3,\omega}(\perp,\perp) \wedge C_2 \wedge C_1) \odot X_2 \mid \{(b_3,b_3+1),(j,2\cdot h+1),(arr(h),v)\} \cup\\
& (C_{3,\omega}(\perp,\perp) \wedge \neg C_2 \wedge C_1) \odot X_2 \mid \{(b_3,b_3+1),(j,2\cdot h),(arr(h),v)\}
\end{aligned}$$

$2\ \not\circlearrowleft$**:**

$$X_2 = \qquad X_1 \mid \{(b_3,b_3(\perp,\omega)),(arr,arr(\perp,\omega)),(h,h(\perp,\omega)),(j,j(\perp,\omega))\}$$

Note that we have collapsed all array assignments into one recursion.

$2 \to$ **exit:** After several simplifications we obtain:

$$\begin{aligned}
X_{\text{exit}} &= (\neg C_{1,\omega}(\bot)) \odot X_1 \mid \\
&\quad \{(b_3, b_3(\bot,\omega)), (arr, arr(\bot,\omega)), (h, (h(\bot,\omega)), (j, j(\bot,\omega))\} \cup \\
&(C_{1,\omega}(\bot) \wedge C_2 \wedge C_{3,\omega}(\bot,\bot)) \odot X_1 \mid \\
&\quad \{(b_3, b_3(\bot,\omega)+1), (arr, arr(\bot,\omega)), (arr(h), v), \\
&\qquad (h, (h(\bot,\omega)), (j, 2\cdot h(\bot,\omega)+1)\} \cup \\
&(C_{1,\omega}(\bot) \wedge \neg C_2 \wedge C_{3,\omega}(\bot,\bot)) \odot X_1 \mid \\
&\quad \{(b_3, b_3(\bot,\omega)+1), (arr, arr(\bot,\omega)), (arr(h), v), \\
&\qquad (h, (h(\bot,\omega)), (j, 2\cdot h(\bot,\omega))\}
\end{aligned}$$

$1 \to$ **exit:**

$$\begin{aligned}
X_{\text{exit}} =& (\neg C_{1,\omega}(\bot)) \odot X_{\text{entry}} \mid \\
&\quad \{(v, arr(k)), (b_3, b_3(\bot,\omega)), (arr, arr(\bot,\omega)), \\
&\qquad (h, (h(k,\omega)), (j, j(\bot,\omega))\} \cup \\
&(C_{1,\omega}(\bot) \wedge C_2 \wedge C_{3,\omega}(\bot,\bot)) \odot X_{\text{entry}} \mid \\
&\quad \{(v, arr(k)), (b_3, b_3(\bot,\omega)+1), (arr, arr(\bot,\omega)), \\
&\qquad (arr(h), v), (h, (h(k,\omega)), (j, 2\cdot h(k,\omega)+1)\} \cup \\
&(C_{1,\omega}(\bot) \wedge \neg C_2 \wedge C_{3,\omega}(\bot,\bot)) \odot X_{\text{entry}} \mid \\
&\quad \{(v, arr(k)), (b_3, b_3(\bot,\omega)+1), (arr, arr(\bot,\omega)), \\
&\qquad (arr(h), v), (h, (h(k,\omega)), (j, 2\cdot h(k,\omega))\}
\end{aligned}$$

entry $\to$ exit: This step is straightforward and is left to the reader.

Now we turn to setting up a recurrence relation for b_3. We obtain for $k \geq 0$

$$b_3(k+1) = b_3(k) + 1 \quad \text{if } (\neg C_3 \wedge C_1).$$

Since C_3 depends on arr, the recursion for b_3 cannot be solved by simple means.

The semantics of discrete loops, however, ensure that no variables except those used in the loop-header affect the number of loop iterations (compare [Bli94] for details). To be more specific, discrete loops are a means to abstract from details of the loop body by declaring that certain parts of the loop body will not affect the timing behavior of the loop. In a sense discrete loops support the oracle in finding conservative and safe estimates of the WCET.

Thus we can concentrate on variable h and the recurrence relation given in the loop header. The theory developed for discrete loops [Bli94] implies that we only have to deal with the following recurrence relation to get a safe upper bound for the number of iterations ($j \geq 1$)

$$\begin{aligned}
h(0) &= k, \\
h(j+1) &= 2 \cdot h(j).
\end{aligned}$$

Its solution is easily determined to be

$$h(j) = k \cdot 2^j.$$

Now, we want to determine ω such that

$$h(\omega) \leq N/2 \leq h(\omega+1).$$

Taking logarithms we get

$$\omega = \lfloor \operatorname{ld} N - \operatorname{ld} k \rfloor - 1$$

and finally we obtain

$$b_3(0,\omega) \leq \lfloor \operatorname{ld} N - \operatorname{ld} k \rfloor - 1 \leq \lceil \operatorname{ld} N \rceil - \lfloor \operatorname{ld} k \rfloor - 1.$$

$$\begin{array}{lcl}
X_{\text{entry}} & = & [\{(arr, \underline{arr}), (N, \underline{N}), (t, \underline{k}), (b_3, 0), (b_6, 0)\}, \text{true}] \\
X_{\text{exit}} & = & (\neg C_5) \odot X_5 \\
X_1 & = & X_{\text{entry}} \mid \{(N, \underline{N}), (k, 1)\} \\
X_2 & = & X_1 \cup X_3 \\
X_3 & = & C_4 \odot X_2 \mid \{(arr, arr'), (b_3, b_3 + 1), (k, k + 1)\} \\
X_4 & = & (\neg C_4) \odot X_2 \mid \{(M, 2)\} \\
X_5 & = & X_4 \cup X_6 \\
X_6 & = & C_5 \odot X_5 \mid \{(t, arr(1)), (arr(1), arr(M)), (arr(M), t), \\
 & & \qquad (arr, arr'), (b_6, b_6 + 1), (M, M + 1)\}
\end{array}$$

TABLE 9. Set of SymEval Equations for Procedure **heapsort**

which is a symbolic formula for the number of iterations in terms of N and k, the parameters of procedure **siftdown**.

This example shows that discrete loops significantly facilitate setting up recurrence relations for the number of loop iterations. In addition, solving this recurrence relations becomes possible because of their comparatively simple structure, which is induced by discrete loops too. Note also that the semantics of discrete loops is easily checked and enforced by a precompiler which produces standard Ada95 code out of Ada code augmented with discrete loops (cf. [Bur96]).

On the other hand, using discrete loops for WCET analysis, some information on the algorithm used in the loop body is lost, but this is certainly out-weighted by the simple recurrence relations produced by discrete loops.

Finally, using Theorem 5.2 we find the following symbolic formula for the upper bound of the timing behavior of procedure **siftdown**:

$$\tau_1 + \tau_2 \cdot (\lceil \operatorname{ld} N \rceil - \lfloor \operatorname{ld} k \rfloor) \tag{6}$$

where $\tau_1, \tau_2 \in T$ are time values but do not denote the timing behavior of some basic blocks.

5.4. **Interprocedural WCET Data-Flow Analysis.** For programs consisting of several procedures and functions, we can build a *call graph*. The nodes of the call graph are the procedures and functions of the program. If procedure A calls B, there is an edge from node A to B in the call graph. If there are no recursive procedures and functions, the call graph is acyclic.

By topologically sorting (cf. [Knu73a, Meh84]) an acyclic call graph, symbolic evaluation can be applied to the graph in such a way that a symbolic formula for the WCET of a procedure P is available before another procedure Q, which calls P, is analyzed.

We have the following theorem:

Theorem 5.4. *If the call graph of a program is acyclic, a WCET estimate for the program can be found by symbolic evaluation of the procedures and functions that constitute the program, provided that a WCET estimate for these procedures and functions can be determined by symbolic evaluation.*

Remark 5.4. WCET analysis of recursive procedures and functions can be performed by the approach described in [BL95, BL96, Bli00].

Remark 5.5. We do not discuss problems originating from aliasing in this paper. How symbolic evaluation can be used to solve these problems is discussed in [BBS99, SBF00].

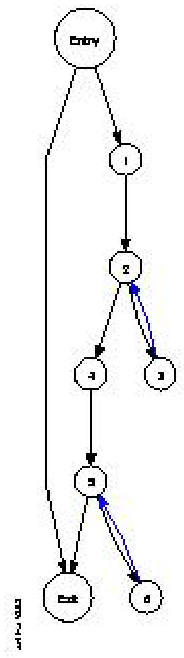

FIGURE 5. CFG of Procedure **heapsort**

Example 3. Returning to our Heapsort example, the CFG of procedure **heapsort** is shown in Figure 5. The SymEval equations can be found in Table 9. We have used the following abbreviations:

$$C_4 = 1 \le k \le N/2,$$
$$C_{4,\omega_1}(k_0) = 1 \le k(k_0,\omega_1) \le N/2,$$
$$C_5 = 2 \le M \le N$$
$$C_{5,\omega_2}(M_0) = 2 \le M(M_0,\omega_2) \le N.$$

Furthermore we have restricted our interest to the instrumentation variables b_3 and b_6. By (arr, arr') we denote that the array arr is assigned a new value by procedure **siftdown**.

According to [Sre95, SGL98] the insertions and loop-breaking operations have to be done as follows.

$6 \to 5$**:**

$$X_5 = X_4 \cup C_5 \odot X_5 \,|\{(t, arr(1)), (arr(1), arr(M)), (arr(M), t) \\ (arr, arr'), (b_6, b_6+1), (M, M+1)\}$$

$5 \not\circlearrowleft$**:**

$$X_5 = X_4 \,| \\ \{(b_6, b_6(\perp,\omega_2)), (t, t(\perp,\omega_2)), (arr, arr(\perp,\omega_2)), (M, M(\perp,\omega_2))\}$$

$5 \to$ **exit:**

$$X_{\text{exit}} = (\neg C_{5,\omega_2}(\perp)) \odot X_4 \,| \\ \{(b_6, b_6(\perp,\omega_2)), (t, t(\perp,\omega_2)), (arr, arr(\perp,\omega_2)), (M, M(\perp,\omega_2))\}$$

$3 \to 2$:

$$X_2 = X_1 \cup C_4 \odot X_2 \mid \{(arr, arr'), (b_3, b_3+1), (k, k+1)\}$$

$4 \to$ exit:

$$\begin{aligned} X_{\text{exit}} = ((\neg C_{5,\omega_2}(\bot)) \wedge (\neg C_{4,\omega_2}(\bot))) \odot X_2 \mid \\ \{(b_6, b_6(\bot,\omega_2)), (t, t(\bot,\omega_2)), (arr, arr(\bot,\omega_2)), (M, M(2,\omega_2))\} \end{aligned}$$

$2 \not\circlearrowleft$:

$$\begin{aligned} X_2 =& \neg C_{4,\omega_1}(\bot) \odot X_1 \mid \\ & \{(b_3, b_3(\bot,\omega_1)), (arr, arr(\bot,\omega_1)), (k, k(\bot,\omega_1))\} \end{aligned}$$

$2 \to$ exit:

$$\begin{aligned} X_{\text{exit}} =& (\neg C_{5,\omega_2}(\bot) \wedge \neg C_{4,\omega_1}(\bot)) \odot X_1 \mid \\ & \{(b_3, b_3(\bot,\omega_1)), (arr, arr(\bot,\omega_1)), (k, k(\bot,\omega_1)), \\ & (b_6, b_6(\bot,\omega_2)), (t, t(\bot,\omega_2)), \\ & (arr, arr(\bot,\omega_2)), (M, M(2,\omega_2))\} \end{aligned}$$

$1 \to$ exit:

$$\begin{aligned} X_{\text{exit}} =& (\neg C_{5,\omega_2}(\bot) \wedge \neg C_{4,\omega_1}(\bot)) \odot X_{\text{entry}} \mid \\ & \{(b_3, b_3(\bot,\omega_1)), (arr, arr(\bot,\omega_1)), (k, k(1,\omega_1)), \\ & (b_6, b_6(\bot,\omega_2)), (t, t(\bot,\omega_2)), \\ & (arr, arr(\bot,\omega_2)), (M, M(2,\omega_2))\} \end{aligned}$$

entry $\to$ exit: This step is straightforward and is left to the reader.

It remains to set up recurrence relations for b_3 and b_6. We obtain

$$b_3(k+1) = b_3(k) + 1 \qquad \text{if } 1 \le k \le N/2$$

and

$$b_6(M+1) = b_3(M) + 1 \qquad \text{if } 2 \le M \le N.$$

It is easy to derive

$$b_3(0,\omega_1) = \lfloor N/2 \rfloor$$

and

$$b_6(0,\omega_2) = N - 1.$$

In order to derive an upper bound for the WCET of the first for-loop, we have to sum equation (6) for $1 \le k \le N/2$. We obtain

$$\begin{aligned} \text{WCET}_{\text{for}_1} \le \sum_{1 \le k \le N/2} \tau_1 + \tau_2 \cdot (\lceil \text{ld}\, N \rceil - \lfloor \text{ld}\, k \rfloor - 1) = \\ \left\lfloor \frac{N}{2} \right\rfloor \tau_1 + \tau_2 \cdot \left(\lfloor N/2 \rfloor \lceil \text{ld}\, N \rceil - \sum_{1 \le k \le N/2} \lfloor \text{ld}\, k \rfloor \right) \end{aligned}$$

which by some manipulations (cf. [GKP89]) gives

$$\text{WCET}_{\text{for}_1} \le \left\lfloor \frac{N}{2} \right\rfloor \tau_1 + \tau_2 \cdot \left(\left\lfloor \frac{5}{2} N \right\rfloor - \lfloor \text{ld}\, N \rfloor - 1 \right). \tag{7}$$

The second for-loop can be treated similarly. Inserting $(k \to 1, N \to M-1)$ into equation (6), we get

$$\tau_1 + \tau_2 \cdot (\lceil \text{ld}(M-1) \rceil). \tag{8}$$

We have to sum (8) for $2 \leq M \leq N$ to obtain an upper bound for the timing behavior of the second for-loop

$$\begin{aligned}\mathrm{WCET}_{\mathrm{for}_2} \leq & \sum_{2 \leq M \leq N} \tau_1 + \tau_2 \cdot (\lceil \mathrm{ld}(M-1) \rceil) = \\ & \sum_{2 \leq M \leq N-1} \tau_1 + \tau_2 \cdot (\lceil \mathrm{ld}\, M \rceil) = \\ & (N-1) \cdot \tau_1 + \tau_2 \cdot \Big((N-1) \lceil \mathrm{ld}(N-1) \rceil - 2^{\lceil \mathrm{ld}(N-1) \rceil} + 1 \Big) .\end{aligned}$$

Adding this to (7) and taking care of some time constant τ_3 for initialization work, we can determine a symbolic formula for the upper bound of the WCET of procedure **heapsort**.

6. Discussion and Related Work

Shortly repeating the major results of this paper we have shown the following:

1. *There exist efficient (almost linear) algorithms for performing worst-case execution time analysis of real-time programs (Theorem 4.2).*
   This is the first result on the timing behavior of WCET analysis in literature.
2. *WCET analysis can be performed on reducible and irreducible CFGs.*
   This, for instance, proves that restrictions concerning goto, break, return, and exit-statements in real-time programming languages are not necessary. In addition, it shows that WCET analysis (even with help of symbolic evaluation) can be applied to assembler or machine-code programs.
3. *Slightly modified elimination methods can be used for solving WCET data-flow frameworks and symbolic evaluation frameworks; iteration algorithms cannot be employed for this purpose.*
4. *Symbolic evaluation can be done for reducible and irreducible CFGs (Observation 2).*
   This shows that symbolic evaluation cannot only be applied to toy languages but it can be used to analyze complex high-order programming language programs as well as low-level assembler or machine-code programs.
5. *Solving (conditional) recurrence relations for symbolic evaluation WCET analysis is performed only on demand.*
   It is not necessary to perform a complete symbolic evaluation of a program in order to determine its WCET. *Symbolic instrumentation* is based on symbolic evaluation but it can be expected to be easier to solve than symbolic evaluation itself.

In the following we compare our approach to other approaches known from literature. Some of the following approaches have already been commented on in Section 1.

1. Tarjan's approach for solving sets of data-flow equations (cf. [Tar81b]) is able to handle WCET data-flow frameworks too. The operators for regular expressions $\cup$, $\cdot$, and $*$ have to be interpreted appropriately for WCET analysis.
   A correct interpretation uses $\cup$ as the meet operator (for example the max meet or the frequency meet operator), $\cdot$ as $+$, which models the juxtaposition of basic blocks, and $*$ as ORACLE(*iter*) for the number of loop iterations.
   Tarjan's method is also well-suited for solving our SymEval framework.
   Nevertheless we have chosen the "loop-breaking rule" approach because it is more intuitive and requires less knowledge on graph theory. In addition, we believe that programmers are more accustomed to thinking in terms of equations than in terms of program paths and regular expressions.

2. The method described in [CBW96] allows WCET analysis depending on the program's input data but focuses on *annotating* the underlying program.

   In contrast our approach relies on symbolic evaluation to extract extraneous information out of the source code in order to perform WCET analysis.

   The *modes* introduced in [CBW96] appear *automatically* in the symbolic WCET formulas of procedures or functions of our approach.
3. The integer linear programming approach of [PS97] is fully oracle-based like our data-flow framework described in Section 4. Like our approach, it is known to produce the exact timing behavior, provided the oracle (programmer) produces exact path execution frequencies and the exact number of iterations of loops. In contrast, our approach presented in Section 4 does not need an oracle for loop iterations.

   In addition, it is not obvious how the approach of [PS97] can be extended to a symbolic evaluation method like that described in Section 5.

   Furthermore ILP is NP-complete and thus has exponential worst-case behavior. Our approach, with the max meet or the frequency meet operator, has almost linear worst-case timing behavior (cf. Theorem 4.2).

7. Conclusion

In this paper we have introduced frameworks based on data-flow equations which provide for estimating the worst-case execution time of real-time programs and for symbolic evaluation of such programs. These frameworks allow several different WCET analysis techniques with various precisions, which range from naïve approaches to exact analysis, provided exact knowledge on the program behavior is available.

We have implemented the algorithm presented in [Sre95, SGL98] for almost all control flow affecting language features of Ada. At the current stage our implementation does not support exceptions. These and tasking features will be implemented in the near future. This implementation allowed us to perform several examples with the max and frequency meet operators. An implementation of the symbolic evaluation data-flow framework is under way and will be completed in the year 2000.

Our approach allows to use off-the-shelf software for manipulating and solving equations. In particular, we are using *Mathematica*[**] for this purpose.

Nevertheless a lot of work remains to be done. The next step will be to study WCET analysis for multi-processor systems and effects of pipelining with help of symbolic evaluation.

References

[AC76] F. E. Allen and J. Cocke, *A program data flow analysis procedure*, Comm. ACM **19** (1976), no. 3, 137–147. 1, 10

[Ada95] ISO/IEC 8652, *Ada reference manual*, 1995. 2, 6

[ALT96] Stephen Alstrup, Peter W. Lauridsen, and Mikkel Thorup, *Dominators in linear time*, Tech. Report TR DIKU 96-35, Department of Computer Science, University of Copenhagen, 1996. 22

[AMWH94] R. Arnold, F. Mueller, D. Whalley, and M. Harmon, *Bounding worst-case instruction cache performance*, Proc. of the Fifteenth IEEE Real-Time Systems Symposium (1994), 172–181. 8

[ASU86] Alfred V. Aho, Ravi Seti, and Jeffrey D. Ullman, *Compilers: principles, techniques, and tools*, Addison-Wesley, Reading, MA, 1986. 1, 5, 19, 22

[**] Mathematica is a trademark of Wolfram Research Inc.

[BB98] Johann Blieberger and Bernd Burgstaller, *Symbolic reaching definitions analysis of Ada programs*, Proceedings of Ada-Europe'98 (Uppsala, Sweden), June 1998, pp. 238–250. 16

[BBS99] Johann Blieberger, Bernd Burgstaller, and Bernhard Scholz, *Interprocedural Symbolic Evaluation of Ada Programs with Aliases*, Ada-Europe'99 International Conference on Reliable Software Technologies (Santander, Spain), June 1999, pp. 136–145. 16, 21, 28

[BBS00] Johann Blieberger, Bernd Burgstaller, and Bernhard Scholz, *Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs*, Ada-Europe'2000 International Conference on Reliable Software Technologies (Potsdam, Germany), June 2000, (to appear). 16

[BFS00] Johann Blieberger, Thomas Fahringer, and Bernhard Scholz, *Symbolic cache analysis for real-time systems*, Real-Time Systems, Special Issue on Worst-Case Execution Time Analysis **18** (2000), no. 2/3, 181–215. 3, 8, 16, 21

[BL95] Johann Blieberger and Roland Lieger, *Real-time recursive procedures*, Proceedings of the 7th EUROMICRO Workshop on Real-Time Systems (Odense), 1995, pp. 229–235. 7, 28

[BL96] Johann Blieberger and Roland Lieger, *Worst-case space and time complexity of recursive procedures*, Real-Time Systems **11** (1996), no. 2, 115–144. 1, 7, 28

[Bli94] Johann Blieberger, *Discrete loops and worst case performance*, Computer Languages **20** (1994), no. 3, 193–212. 1, 20, 25, 27, 27

[Bli00] Johann Blieberger, *Real-time properties of indirect recursive procedures*, Information and Computation (2000), (to appear). 1, 7, 28

[Bur96] Bernd Burgstaller, *The WOOP preprocessor – an implementation of discrete loops*, Diploma thesis, TU Vienna, Dept. of Automation, 1996. 28

[CBW96] Roderick Chapman, Alan Burns, and Andy Wellings, *Combining static worst-case timing analysis and program proof*, Real-Time Systems **11** (1996), no. 2, 145–171. 1, 2, 2, 32, 32

[CHT79] Thomas E. Cheatham, Glenn H. Holloway, and Judy A. Townley, *Symbolic evaluation and the analysis of programs*, IEEE Trans. on Software Engineering **5** (1979), no. 4, 403–417. 16, 16

[CJM$^+$92] B.A. Carré, T.J. Jennings, F.J. Maclennan, P.F. Farrow, and J.R. Garnsworthy, *SPARK: The SPADE Ada kernel*, Program Validation Ltd., 3.1 ed., 1992. 2

[CR81] L.A. Clarke and D.J. Richardson, *Symbolic evaluation methods for program analysis*, Program Flow Analysis (S.S. Muchnik and N.D. Jones, eds.), Prentice Hall, Englewood Cliffs, New Jersey, 1981, pp. 264–300. 16

[Deu94] Alain Deutsch, *Interprocedural may-alias analysis for pointers: Beyond k-limiting*, Proceedings of the ACM SIGPLAN'94 Conf. on PLDI, 1994, pp. 230–241. 21

[FS97] T. Fahringer and B. Scholz, *Symbolic Evaluation for Parallelizing Compilers*, Proc. of the ACM International Conference on Supercomputing, July 1997. 16, 16

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, *Concrete mathematics*, Addison-Wesley, Reading, MA, 1989. 13, 15, 30

[GR91] Narain Gehani and Krithi Ramamritham, *Real-time Concurrent C: A language for programming dynamic real-time systems*, The Journal of Real-Time Systems **3** (1991), 377–405. 2, 7, 21

[GW76] Susan L. Graham and Mark Wegman, *Fast and usually linear algorithm for global flow analysis*, J. ACM **23** (1976), no. 1, 172–202. 1, 10

[HBW94] Marion G. Harmon, T. P. Baker, and David B. Whalley, *A retargetable technique for predicting execution time of code segments*, Real-Time Systems **7** (1994), 159–182. 8

[Hoa69] Charles Anthony Richard Hoare, *An axiomatic basis for computer programming*, Communications of ACM **12** (1969), 576–580. 2

[HS91] Wolfgang A. Halang and Alexander D. Stoyenko, *Constructing predictable real time systems*, Kluwer Academic Publishers, Boston, 1991. 1, 1, 1, 7, 21

[HU77] Matthew S. Hecht and Jeffrey D. Ullman, *A simple algorithm for global data flow analysis problems*, SIAM J. Comput. **4** (1977), no. 4, 519–532. 1, 10

[HWH95] C. A. Healy, D. B. Whalley, and M. G. Harmon, *Integrating the timing analysis of pipelining and instruction caching*, Proc. of the Sixteenth IEEE Real-Time Systems Symposium (1995), 288–297. 8

[ITM90] Yutaka Ishikawa, Hideyuki Tokuda, and Clifford W. Mercer, *Object-oriented real-time language design: Constructs for timing constraints*, ECOOP/OOPSLA '90 Proceedings, October 1990, pp. 289–298. 1, 7, 21

[Kil73] G. Kildall, *A unified approach to global program optimization* , Proc. of the First ACM Symposium on Principles of Programming Languages (New York, NY), 1973, pp. 194–206. 6

[Knu73a] Donald E. Knuth, *Fundamental algorithms*, second ed., The Art of Computer Programming, vol. 1, Addison-Wesley, Reading, Mass., 1973. 28

[Knu73b] Donald E. Knuth, *Sorting and searching*, The Art of Computer Programming, vol. 3, Addison-Wesley, Reading, Mass., 1973. 25

[KS86] Eugene Kligerman and Alexander D. Stoyenko, *Real-time Euclid: A language for reliable real-time systems*, IEEE Transactions on Software Engineering **12** (1986), no. 9, 941–949. 1

[KU76] John B. Kam and Jeffrey D. Ullman, *Global data flow analysis and iterative algorithms*, J. ACM **23** (1976), no. 1, 158–171. 1

[KU77] John B. Kam and Jeffrey D. Ullman, *Monotone data flow analysis frameworks*, Acta Informatica **7** (1977), 305–317. 1

[Lan92] William Landi, *Undecidability of static analysis*, Lett. Prog. Lang. Syst. **1** (1992), no. 4, 323–337. 21

[Li92] Zhiyuan Li, *Array privatization for parallel execution of loops*, Proceedings of the International Conference on Supercomputing, 1992, pp. 313–322. 21

[LL94] J.C. Liu and H.J. Lee, *Deterministic upperbounds of the worst-case execution times of cached programs*, Proc. of the Fifteenth IEEE Realt-Time Systems Symposium (1994), 182–191. 8

[LM95] Yau-Tsun Steven Li and Sharad Malik, *Performance analysis of embedded software using implicit path enumeration*, ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (La Jolla, California), vol. 30, ACM SIGPLAN Notices, June 1995. 2

[LR92] William Landi and Barbara G. Ryder, *A safe approximate algorithm for interprocedural pointer aliasing*, Proceedings of the ACM SIGPLAN'92 PLDI-6, 1992, pp. 235–248. 21

[MACT89] Aloysius K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat, *Evaluating tight execution time bounds of programs by annotations*, Proc. IEEE Workshop on Real-Time Operating Systems and Software, 1989, pp. 74–80. 2

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam, *Array-data flow analysis and its use in array privatization*, Proceedings of the 20th ACM POPL'93, 1993, pp. 2–15. 21

[Meh84] Kurt Mehlhorn, *Graph algorithms and NP-completeness*, Data Structures and Algorithms, vol. 2, Springer-Verlag, Berlin, 1984. 28

[MR90] Thomas J. Marlowe and Barbara G. Ryder, *Properties of data flow frameworks – a unified model*, Acta Informatica **28** (1990), 121–163. 1, 4, 5

[NP93] Vivek Nirkhe and William Pugh, *A partial evaluator for the Maruti hard real-time system*, The Journal of Real-Time Systems **5** (1993), 13–30. 1, 2

[Par93] Chang Yun Park, *Predicting program execution times by analyzing static and dynamic program paths*, The Journal of Real-Time Systems **5** (1993), 31–62. 1, 2

[Pau88] Marvin C. Paull, *Algorithm design – a recursion transformation framework*, Wiley Interscience, New York, NY, 1988. 9, 9, 10

[PK89] Peter Puschner and Christian Koza, *Calculating the maximum execution time of real-time programs*, The Journal of Real-Time Systems **1** (1989), 159–176. 1, 2, 7, 21

[Plo80] Erhard Ploedereder, *A semantic model for the analysis and verification of programs in general, higher-level languages*, Ph.D. thesis, Division of Applied Sciences, Harvard University, 1980. 16

[PS97] Peter Puschner and Anton V. Schedl, *Computing maximum task execution times – a graph-based approach*, Real-Time Systems **13** (1997), no. 1, 67–91. 1, 2, 2, 32, 32

[Ram94] G. Ramalingam, *The undecidability of aliasing*, ACM Transactions on Programing Languages and Systems **16** (1994), no. 5, 1467–1471. 21

[Ram96] G. Ramalingam, *Data Flow Frequency Analysis*, PLDI'96, May 1996, pp. 267–277. 13

[Rog92] Hartley Rogers, *Theory of recursive functions and effective computability*, MIT Press, Cambridge, MA, 1992. 20

[RP86] Barbara G. Ryder and Marvin C. Paull, *Elimination algorithms for data flow analysis*, ACM Computing Surveys **18** (1986), no. 3, 277–316. 1, 10

[SBF00] Bernhard Scholz, Johann Blieberger, and Thomas Fahringer, *Symbolic Pointer Analysis for Detecting Memory Leaks*, ACM SIGPLAN Workshop on "Partial Evaluation and Semantics-Based Program Manipulation" (PEPM'00) (Boston), January 2000. 21, 28

[Sch96] Bernhard Scholz, *Symbolische Verifikation von Echtzeitprogrammen*, Diploma thesis, TU Vienna, Dept. of Automation, 1996. 16

[Sed88] Robert Sedgewick, *Algorithms*, second ed., Addison-Wesley, Reading, MA, 1988. 25

[SGL98] Vugranam C Sreedhar, Guang R. Gao, and Yong-Fong Lee, *A new framework for elimination-based data flow analysis using dj graphs*, ACM Transactions on Programming Languages and Systems **20** (1998), no. 2, 388–435. 1, 10, 10, 12, 13, 14, 19, 20, 23, 26, 29, 32

[Sha89] Alan C. Shaw, *Reasoning about time in higher-level language software*, IEEE Transactions on Software Engineering **15** (1989), no. 7, 875–889. 1, 2, 2, 7, 21

[SHH91] Alexander Stoyenko, V. Hamacher, and R. Holt, *Analyzing hard real-time programs for guaranteed schedulability*, IEEE Transactions on Software Engineering **17** (1991), no. 8, 737–750. 1

[Sre95] Vugranam C. Sreedhar, *Efficient program analysis using DJ graphs*, Ph.D. thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, 1995. 1, 10, 10, 12, 13, 14, 19, 20, 23, 26, 29, 32

[SS93] Russel Schaffer and Robert Sedgewick, *The analysis of heapsort*, Journal of Algorithms **15** (1993), 76–100. 25

[Tar81a] Robert Endre Tarjan, *Fast algorithms for solving path problems*, J. ACM **28** (1981), no. 3, 594–614. 1, 10, 14

[Tar81b] Robert Endre Tarjan, *A unified approach to path problems*, J. ACM **28** (1981), no. 3, 577–593. 1, 2, 31

Department of Automation (183/1), Technical University of Vienna, Treitlstr. 1/4, A-1040 Vienna

E-mail address: `blieb@auto.tuwien.ac.at`