Programming Language Support for Estimating the Worst-Case Performance of Object-Oriented Real-Time Programs*

Johann Blieberger
Department of Automation
Technical University of Vienna, Austria
Johann.Blieberger@tuwien.ac.at

Abstract: The purpose of this paper is to survey programming language support for estimating the worst-case performance of object-oriented real-time programs. We present several solutions offered by existing (object-oriented) real-time languages and results of our own research. In detail, we introduce a new kind of loops, so-called discrete loops, we show that recursion can be used for real-time applications without harm, if a few conditions are met, and we present an approach how the interface of a class can be augmented by information concerning the worst-case time and space behavior.

Key Words: object-oriented real-time systems, worst-case performance, loops, recursions

Category: J.7, D.4.7, C.3, D.3, D.3.2, D.3.3

1 Introduction

The most significant difference between real-time systems and other computer systems is that not only the system behavior has to be correct but that the result of a computation must be available within a predefined deadline. It has turned out that major progress in order to guarantee the timeliness of real-time systems can only be achieved if the *scheduling problem* is solved properly. Most scheduling algorithms assume that the runtime of a task is known a priori (see e.g. [Liu and Layland, 1973, Halang and Stoyenko, 1991, Mok, 1984]). Thus the *worst-case performance* of a task plays a crucial role. Many other terms are used in literature, e.g. worst-case execution time. In the context of programming languages, however, we prefer worst-case performance.

Mainly the notions of *time* and *scheduling* are reflected in real-time programming languages. Programming language support for estimating the worst-case performance of real-time programs has not been considered that important. In fact, usually real-time programming languages *restrict* ordinary language constructs to facilitate estimating worst-case performance. The purpose of this paper is to show that *augmenting* programming languages in order to ease estimating worst-case performance is possible and results in great improvements for real-time applications. Besides, some of the language features presented in this paper are of interest for non-real-time applications as well, because they greatly improve testability, reusability, and maintenability of programs.

* Supported by the Austrian Science Foundation (FWF) under grant P10188-MAT.

The most difficult tasks in estimating the timing behavior of a program are to determine the *number of iterations of loops* and to handle problems originating from the use of *recursion*.

We will survey several approaches to these problems in this paper and we will present several solutions offered by existing (object-oriented) real-time languages and results of our own research. In Section 2 loops will be discussed, in Section 3 real-time recursion will be treated.

In addition, we are interested in how *timing information* provided at an object's interface can be used to estimate the overall timing behavior of real-time programs and most important how this timing information is presented. Until now object-oriented real-time systems have only been studied in a very restricted manner. Our treatment can be found in Section 4.

Remark. In this paper we will use the following notations.

- By $\log N = \log_e N$ we denote the natural logarithm of N.
- By $\mathrm{ld}\ N$ we denote the binary logarithm of N.
- By $\log_a N = \frac{\log N}{\log a}$ we denote the logarithm to the base a.
- The greatest integer $n \leq x$ is denoted by $\lfloor x \rfloor$.
- The smallest integer $n \geq x$ is denoted by $\lceil x \rceil$.
- By $\Delta f(x) := f(x+1) - f(x)$ we denote the difference operator of finite calculus.

2 Loops

2.1 Introduction

Ordinary programming languages support two different forms of loop-statements:

for-loops: A loop variable assumes all values of a given integer range. Starting with the smallest value of the range, the loop-body is iterated until the value of the loop variable is outside the given range.
Some programming languages allow for starting with the largest value and decrementing the loop variable, others allow for defining a fixed step by which the loop variable is incremented or decremented.

general loops: The other loop-statement is of a very general form and is intended for implementing those loops that can not be handled by for-loops. These loops include while-loops, repeat-loops, and loops with exit-statements (see e.g. [Ada, 1995]).

Determining the number of iterations of a for-loop is trivial. For example the loop-body of the loop

```
for i in 1..N loop
   -- loop body
end loop;
```

is performed exactly N times. Even nested loops do not constitute any problem.

General loops, however, represent a very difficult task. In order to estimate the worst-case performance of general loops, many methods and tools have been developed, e.g. [Puschner and Koza, 1989, Park, 1993, Nirkhe and Pugh, 1993, Halang and Stoyenko, 1991]. In the following we will discuss some of them.

2.2 Recent Approaches

In this subsection we present recent approaches to the problem of determining the number of iterations of general loops.

- In [Puschner and Koza, 1989] language constructs have been introduced in order to let the programmer integrate knowledge about the actual behavior of algorithms which can not 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 can not 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.
- In [Halang and Stoyenko, 1991] the programming language Real-Time Euclid and a corresponding *schedulability analyzer* are described. The estimation of worst-case performance is facilitated by restricting language constructs, e.g. *constant loop bounds* are required.
- *Partial evaluation* is used in [Nirkhe and Pugh, 1993] 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 needs not 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.
- The idea to estimate worst-case performance of programs written in higher-level languages has been introduced in [Shaw, 1989]. So-called *schemas* are used to estimate the best and worst-case performance of statements of higher-level languages and an extension of Hoare logic (see [Hoare, 1969]) 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.
- Continuing and extending [Shaw, 1989] best and worst-case performance is estimated by employing static and dynamic program paths analysis in [Park, 1993]. 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.
  The reported examples (see [Park, 1993]) show that tight bounds can be derived by means of this method. On the other hand, the user must specify upper bounds for general loop statements.
- Determining the execution time of a code segment is also mentioned in [Gehani and Ramamritham, 1991]. Real-time concurrent C uses a tool which originally is based on [Mok et al., 1989]. The code is allowed to have loops with user-specified loop bounds.
- In [Li and Malik, 1995] *integer linear programming* is used to find the best and worst-case execution time of programs. (A similar approach has been

persued in the MARS project (see [Schedl, 1993]).) The programmer has to supply lower and upper bounds for the number of loop iterations.

Summing up, most researchers try to ease the task of estimating the number of general loop iterations by *forbidding* general loops, i.e., by forcing the user to supply constant upper bounds for the number of iterations. Another approach is to let the user specify a time bound within which the loop has to complete (see e.g. [Ishikawa et al., 1990]). In any case the user, i.e., the programmer, has to react to such exceptional cases.

In the following subsections we present *discrete loops*, which have been introduced in [Blieberger, 1994].

2.3 Introduction to Discrete Loops

In contrast to for-loops, discrete loops allow for a more complex dependency between two successive values of the loop variable. In fact an arbitrary functional dependency between two successive values of the loop variable is admissible, but this dependency must be constrained in order to ensure that the loop completes and to determine the number of iterations of the loop. Details of this constraints will follow below.

Like for-loops discrete loops have a loop variable and an integer range associated with them. (In Section 2.7 a more general form of discrete loops is introduced which does not need a discrete range, but we defer a thorough discussion of these loops until then.) The fact that the loop is allowed to range over discrete values, coined the name *discrete loop*. The major difference to for-loops is that the loop variable is not assigned each of the values of the range. Which values are assigned to the loop variable, is completely governed by the loop-body. The loop-header, however, contains a list of all those values that can possibly be assigned to the loop variable during the next iteration. In fact each item of this list of values is a function of the loop variable.

A simple example is shown in Figure 1. In this example the loop variable `k` will assume the values $1, 2, 4, 8, 16, 32, 64, \ldots$ until finally a value greater than `N` would be reached. Of course the effect of this example can also be achieved by a simple for-loop, where the powers of two are computed within the loop body.

```
discrete k in 1..N new k := 2*k loop
   -- loop body
end loop;
```

Figure 1: A simple example of a discrete loop

A more complex example is depicted in Figure 2. In this example the loop variable `k` can assume the values $1, 2, 4, 9, 18, 37, 75, \ldots$ until finally a value greater than `N` would be reached. But it is also possible that `k` follows the sequence $1, 3, 6, 13, 26, 52, 105, \ldots$. Here the same effect can not be achieved by a for-loop, because the value of the loop variable can not be determined exactly before the loop body has been completely elaborated. The reason for this is the *indeterminism* involved in discrete loops.

```
discrete k in 1..N new k := 2*k | 2*k+1 loop
  -- loop body
end loop;
```

Figure 2: A more complex example of a discrete loop

The term "indeterminism" requires some explanation: Clearly the loop body *determines* exactly which of the given alternatives is chosen, thus one can say that there definitely is no indeterminism involved. On the other hand, from an outside-view of the loop one can not determine which of the alternatives will be chosen, without having a closer look at the loop body or without exactly knowing which data are processed by the loop. It is this "outside-view" indeterminism we mean here. Furthermore this indeterminism enables us to estimate the number of loop iterations quite accurately without having to know all details of the loop body. Thus discrete loops ease estimating the worst-case performance of real-time programs.

By the way, a loop like that in Figure 2 occurs in a not-recursive implementation of *Heapsort* (see [Knuth, 1973b] or [Schaffer and Sedgewick, 1993] for a more readable form in a high-order programming language). Sections 2.5.2 and 2.8.1 will be concerned with algorithms that can profit from discrete loops; Heapsort will be treated in detail in Section 2.5.2.

There are two main reasons for stating this functional dependency between successive values of the loop variable in the loop-header:

1. The compiler or, if it can not be done statically at compile-time, the runtime system should check if the loop variable does in fact obtain one of the possible values stated in the loop-header. This will evidently ease debugging and shift some runtime errors to compile-time errors. In fact, if the information given in the loop-header is incorrect, this results in a *programming error*, not in a *timing error*. Of course this programming error could cause a timing error.
2. Under some circumstances, the information in the loop-header will make determining the number of loop iterations feasible.

2.4 Theoretical Treatment of Discrete Loops

In the following we will not give proofs of lemmas and theorems. The interested reader can find them in [Blieberger, 1994].

Discrete loops can be defined using a range of any discrete type, e.g. an enumeration. In our theoretical treatment, however, we will assume that the range is `1..N` and that the loop variable starts with $k_1 = s$, where s is the starting value of the loop. This restriction, however, does not inhibit transferring our results to the cases mentioned above. If s is not in the range `1..N`, the loop-body is not executed, rather the control-flow of the program is transferred to the first statement after the loop.

Definition 1. A *discrete loop* is characterized by $N \in \mathbf{N}$ and a finite number of functions $f_i : \mathbf{N} \to \mathbf{N}$, $1 \leq i \leq e$.

Definition 2. An *iteration sequence* (k_ν) is defined by the recurrence relation

$$k_1 := s, \quad s \in [1, N]$$
$$k_{\nu+1} := f_i(k_\nu)$$

for some i. The set of all possible iteration sequences is denoted by $\mathcal{K} = \{(k_\nu)\}$.

Remark. Note that $k_\nu \in \mathbf{N}$ for all $\nu \in \mathbf{N}$.

Definition 3. An iteration sequence (k_ν) is said to *complete* if $1 \leq k_\nu \leq N$ for all $\nu \leq \omega$, but $k_{\omega+1} < 1$ or $k_{\omega+1} > N$ for some $\omega \in \mathbf{N}$. The number ω is denoted by len k_ν and called the length of (k_ν). It corresponds to the number of iterations of the discrete loop if the loop variable iterates through (k_ν).

Definition 4. A discrete loop is called a *completing discrete loop* if all $(k_\nu) \in \mathcal{K}$ are completing sequences for all N and for all $s \in [1, N]$.

Definition 5. Let a discrete loop be characterized by N and the iteration functions $f_i(x)$, $1 \leq i \leq e$. Let the initial value of the loop variable be s. For $1 \leq x \leq N$ associating to each function f_i a function $\overline{f}_i$ by

$$\overline{f}_i(x) = \begin{cases} 0, & \text{if } f_i(x) < 1, \\ f_i(x), & \text{if } 1 \leq f_i(x) \leq N, \text{ and} \\ N+1, & \text{if } f_i(x) > N, \end{cases}$$

we define the corresponding *loop digraph* $\mathcal{L}$ by the set of vertices V which is defined by

$$V = \{0, 1, \ldots, N, N+1\}$$

and the set of edges E, where E is defined by

$$(v, w) \in E \quad \Leftrightarrow \quad w = \overline{f}_i(v)$$

for some $i \in [1, e]$.

With these definitions the following lemma is trivially true.

Lemma 6. *A discrete loop completes if and only if the corresponding loop digraph is acyclic.*

Each acyclic digraph can be *topologically sorted* (see [Mehlhorn, 1984a]), i.e., we can find a mapping ord : $V \to \{0, 1, \ldots, N, N+1\}$ such that for all edges $(v, w) \in E$ we have ord$(v) <$ ord(w). Since we are only interested in completing discrete loops, we restrict ourselves to discrete loops that result in topologically sorted loop digraphs. This is certainly the case if $f_i(x) > x$ or if $f_i(x) < x$ for all $x \in \mathbf{N}$ and for all $i \in [1, e]$. The next section is devoted to such loops.

2.5 Monotonical Discrete Loops

Definition 7. A sequence (k_ν) is called *strictly monotonically increasing* if for all $\nu \geq 1$, we have $k_{\nu+1} > k_\nu$. It is called *strictly monotonically decreasing* if $k_{\nu+1} < k_\nu$ for all $\nu \geq 1$.

Definition 8. A discrete loop is called a *monotonically increasing discrete loop* if all $(k_\nu) \in \mathcal{K}$ are strictly monotonically increasing sequences. It is called a *monotonically decreasing discrete loop* if all $(k_\nu) \in \mathcal{K}$ are strictly monotonically decreasing sequences. A discrete loop is called a *monotonical discrete loop* if it is either monotonically increasing or monotonically decreasing.

Lemma 9. *A monotonical discrete loop is completing.*

Lemma 10. *Let a monotonically increasing discrete loop be characterized by N and the functions f_i. Then all functions f_i fulfill*

$$f_i(x) > x$$

for all $x \in [1, N]$.

Lemma 11. *Let a monotonically decreasing discrete loop be characterized by N and the functions f_i. Then all functions f_i fulfill*

$$f_i(x) < x$$

for all $x \in [1, N]$.

2.5.1 Syntactical and Semantical Issues of Monotonical Discrete Loops

Although the syntax of discrete loops is certainly important, we consider the semantical issues more important. In order to be able to demonstrate the advantages of discrete loops over conventional loops, however, we define an Ada-like syntax which will be used in the following examples. But it is important to note that an appropriate syntax can be defined for other languages too.

The syntax of a monotonical discrete loop is given by a notation similar to that in [Ada, 1995].

```
loop_statement ::=
    [loop_simple_name:]
        [iteration_scheme] loop
            sequence_of_statements
        end loop [loop_simple_name];

iteration_scheme ::= while condition
    | for for_loop_parameter_specification
    | discrete discrete_loop_parameter_specification

for_loop_parameter_specification ::=
    identifier in [reverse] discrete_range

discrete_loop_parameter_specification ::=
```

identifier := initial_value **in** [**reverse**] discrete_range
new identifier := list_of_iteration_functions

list_of_iteration_functions ::=
iteration_function { | iteration_function }

iteration_function ::= expression

For a loop with a **discrete** iteration scheme, the loop parameter specification is the declaration of the *loop variable* with the given identifier. The loop variable is an object whose type is the base type of the discrete range. The initial value of the loop variable is given by initial_value. The optional keyword **reverse** defines the loop to be monotonically decreasing; if it is missing the loop is considered to be monotonically increasing. Within the sequence of statements the loop variable behaves like any other variable, i.e. for example, it can be used on both sides of an assignment statement.

Before the sequence of statements is executed, the list of iteration functions is evaluated. This results in a list of *possible successive values*. It is also checked whether all of these values are greater than the value of the loop variable if the keyword **reverse** is missing, or whether they are smaller than the value of the loop variable if **reverse** is present. If one of these checks fails, the exception **monotonic_error** is raised.

After the sequence of statements has been executed, it is checked whether the value of the loop variable is contained in the list of possible successive values. If this check fails, the exception **successor_error** is raised.

If the value of the loop variable is still within the discrete range stated in the loop header, the loop is iterated (at least) once more. If it is not within the range, the loop completes.

Remark. The semantics of monotonical discrete loops ensure that such a loop will always complete, either because the value of the loop variable is outside the given discrete range or because one of the above checks fails, i.e., one of the exceptions **monotonic_error** or **successor_error** is raised.

Remark. A corresponding compiler is free to perform as many checks as it likes in order to inhibit one of the runtime exceptions **monotonic_error** and **successor_error**. This can be done by ensuring that the iteration functions are monotonical functions and by performing data-flow analysis to make sure that **successor_error** will never be raised. Thus a lot of runtime checks can be avoided.

Moreover the compiler might even detect the number of iterations of the loop, which is a valuable result for real-time applications. Clearly the number of iterations depends on the initial value of the loop variable, on the discrete range (especially the number of elements in the range), and on the iteration functions.

2.5.2 Some Examples of Monotonical Discrete Loops

Heapsort

An Ada-like implementation of Heapsort using a monotonical discrete loop is shown in Figure 3. Referring to the code shown in Figure 3, we easily see that

```
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);
         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);
   end loop;
end heapsort;
```

Figure 3: Implementation of Heapsort using a Discrete Loop

the number of iterations of the discrete loop in procedure `siftdown` is bounded above by the length of $(h_\nu^{(\min)})$, where $(h_\nu^{(\min)})$ fulfills the recurrence relation

$$\begin{aligned} h_1^{(\min)} &= k \\ h_{\nu+1}^{(\min)} &= 2h_\nu^{(\min)} \end{aligned} \tag{1}$$

since the length of any loop sequence containing two successive elements that fulfill $k_{\nu+1} = 2k_\nu + 1$ will be smaller than that of $(h_\nu^{(\min)})$. (How lower and

upper bounds of the number of iterations of discrete loops can be estimated, is investigated in detail in Section 2.6.)

Solving (1) we arrive at

$$h_\nu^{(\min)} = k2^{\nu-1}.$$

We want to determine the value of ω such that

$$h_\omega^{(\min)} \le N/2 \le h_{\omega+1}^{(\min)},$$

thus taking logarithms we obtain

$$\omega = \lfloor \mathrm{ld}\ N - \mathrm{ld}\ k \rfloor \le \lceil \mathrm{ld}\ N \rceil - \lfloor \mathrm{ld}\ k \rfloor$$

for the number of iterations of the discrete loop in procedure `siftdown`.

The number F of iterations of the first for-loop in the main procedure is bounded above by

$$F \le \sum_{k=1}^{\lfloor N/2 \rfloor} \lceil \mathrm{ld}\ N \rceil - \lfloor \mathrm{ld}\ k \rfloor = \lfloor N/2 \rfloor \lceil \mathrm{ld}\ N \rceil - \sum_{k=1}^{\lfloor N/2 \rfloor} \lfloor \mathrm{ld}\ k \rfloor.$$

Using (see [Graham et al., 1989, problem 3.34])

$$\sum_{k=1}^{n} \lceil \mathrm{ld}\ k \rceil = n\lceil \mathrm{ld}\ n \rceil - 2^{\lceil \mathrm{ld}\ n \rceil} + 1 \tag{2}$$

and noticing that

$$\lceil \mathrm{ld}\ k \rceil - \lfloor \mathrm{ld}\ k \rfloor = \begin{cases} 0 \text{ if } k \text{ is a power of 2, and} \\ 1 \text{ otherwise,} \end{cases}$$

we obtain

$$\sum_{k=1}^{n} \lfloor \mathrm{ld}\ k \rfloor = n\lceil \mathrm{ld}\ n \rceil - n - 2^{\lceil \mathrm{ld}\ n \rceil} + \lfloor \mathrm{ld}\ n \rfloor + 2. \tag{3}$$

Hence

$$-\sum_{k=1}^{\lfloor N/2 \rfloor} \lfloor \mathrm{ld}\ k \rfloor = -\lfloor N/2 \rfloor \lceil \mathrm{ld}\ \lfloor N/2 \rfloor \rceil + \lfloor N/2 \rfloor + 2^{\lceil \mathrm{ld}\ \lfloor N/2 \rfloor \rceil} - \lfloor \mathrm{ld}\ \lfloor N/2 \rfloor \rfloor - 2.$$

Furthermore we have (see [Graham et al., 1989, problem 3.19])

$$\lceil \mathrm{ld}\ \lfloor N/2 \rfloor \rceil \ge \lfloor \mathrm{ld}\ \lfloor N/2 \rfloor \rfloor = \lfloor \mathrm{ld}\ (N/2) \rfloor = \lfloor \mathrm{ld}\ N \rfloor - 1$$

and

$$\lceil \mathrm{ld}\ \lfloor N/2 \rfloor \rceil \le \lfloor \mathrm{ld}\ \lfloor N/2 \rfloor \rfloor + 1 = \lfloor \mathrm{ld}\ N \rfloor.$$

Thus

$$F \le \lfloor N/2 \rfloor (\lceil \mathrm{ld}\ N \rceil - \lfloor \mathrm{ld}\ N \rfloor + 2) + 2^{\lfloor \mathrm{ld}\ N \rfloor} - \lfloor \mathrm{ld}\ N \rfloor - 1 \le$$
$$3\lfloor N/2 \rfloor + 2^{\mathrm{ld}\ N} - \lfloor \mathrm{ld}\ N \rfloor - 1 \le \lfloor \tfrac{5}{2} N \rfloor - \lfloor \mathrm{ld}\ N \rfloor - 1.$$

Remark. Note that F is almost linear in N although we have a discrete loop "nested" within a for-loop.

The number L of iterations of the second for-loop in the main procedure is bounded above by

$$L \leq \sum_{t=2}^{N} \lfloor \mathrm{ld}\ t \rfloor .$$

Using (3), L can be estimated by

$$L \leq N\lceil \mathrm{ld}\ N \rceil - N - 2^{\lceil \mathrm{ld}\ N \rceil} + \lfloor \mathrm{ld}\ N \rfloor + 2.$$

In a very similar way a lower bound for the number of iterations can be found.

These computations are very easy and we think that they can also be performed by an automated tool during compile-time. Section 2.6 contains some theoretical foundations in order to ease the task of these compile-time computations.

Concluding we would like to remark that the purpose of this section was not to show how to analyze Heapsort. In fact, the worst-case timing behavior (see [Knuth, 1973b]) and even the average timing behavior of Heapsort (see [Schaffer and Sedgewick, 1993]) are well understood. The purpose of this section was to show that monotonical discrete loops can ease the task of worst-case timing analysis of algorithms significantly. Sometimes the analysis is so easy that it can be performed by an automated tool.

Other Examples

Other examples showing the advantages of discrete loops over general while or repeat-loops include the bottom-up version of *Mergesort* (see [Sedgewick, 1988]), the solution of *Josephus' Problem* (see [Graham et al., 1989]), and *Euclid's algorithm*.

2.6 The Number of Iterations of a Monotonical Discrete Loop

Because of the indeterminism involved in the definition of discrete loops, the number of iterations of such a loop cannot be determined exactly. We can, however, find lower and upper bounds for the number of iterations. Corresponding theoretical results are given in the following subsection.

2.6.1 Lower and Upper Bounds

Definition 12. Let $\omega(\mathcal{K})$ denote the multi-set of the length of all sequences $(k_\nu) \in \mathcal{K}$ of a monotonical discrete loop and let

$$\overline{\mathbf{l}} = \min \omega(\mathcal{K}) \quad \text{and} \quad \overline{\mathbf{u}} = \max \omega(\mathcal{K})$$

denote the lower and upper bound of the length of the sequences. These represent lower and upper bounds for the number of iterations of the discrete loop too.

In the rest of this section we will only be concerned with montonically increasing discrete loops. Of course the following treatment can easily be modified in order to deal with monotonically decreasing discrete loops.

The loop digraph corresponding to a certain loop is very important in this section to prove properties of discrete loops. Note, however, that the vertex 0 can be avoided since the underlying loop is monotonically increasing.

In order to calculate $\overline{\mathbf{u}}$ we can use an algorithm given in [Mehlhorn, 1984a] which determines the longest path in topologically sorted digraphs. The path is supposed to start at node s.

```
for k in 1..N+1 loop
  c(k) := -∞
end loop;
c(s) := 0;
for k in s..N loop
  for i in 1..e loop
    c(f̄_i(k)) := max{c(f̄_i(k)), c(k)+1};
  end loop;
end loop;
```

A similar procedure can be used to determine the shortest path in $\mathcal{L}$.

```
for k in 1..N+1 loop
  c(k) := +∞
end loop;
c(s) := 0;
for k in s..N loop
  for i in 1..e loop
    c(f̄_i(k)) := min{c(f̄_i(k)), c(k)+1};
  end loop;
end loop;
```

Remark. The shortest and longest path, i.e., the final value of $c(N+1)$, computed by the algorithms above, correspond to $\overline{\mathbf{l}}$ and $\overline{\mathbf{u}}$, respectively.

Summing up, we have found algorithms that compute lower and upper bounds of the number of iterations of monotonical discrete loops in time $O(Ne)$. The following Theorem 14 will show that under certain conditions $\overline{\mathbf{u}}$ and $\overline{\mathbf{l}}$ can be determined much easier.

Definition 13. Let a monotonically increasing discrete loop be given by the number N and the iteration functions $f_i(x)$. Then we denote by

$$k_{\nu+1}^{(\min)} = \min_i f_i(k_\nu^{(\min)}) \quad \text{and by} \quad k_{\nu+1}^{(\max)} = \max_i f_i(k_\nu^{(\max)})$$

the sequences that always assume the smallest and largest possible values, respectively.

Theorem 14. *If for all $1 \le i \le e$ $f_i(1) > 1$ and $\Delta f_i(x) \ge 1$ for all $x \in \boldsymbol{N}$, then*

1. *the corresponding discrete loop completes,*
2. *the length of (k_ν^{max}) is equal to $\overline{\mathbf{l}}$, and*

3. the length of (k_ν^{min}) *is equal to* $\overline{u}$.

Remark. Theorem 14 may seem trivial, but it is not (for details see the proof in [Blieberger, 1994]).

If $f_{\min}(x) = \min_i\{f_i(x)\}$ and $f_{\max}(x) = \max_i\{f_i(x)\}$ can be determined independently of x, Theorem 14 enables us to restrict our interest to two single functions in estimating lower and upper bounds of the number of iterations of a discrete loop.

2.6.2 Some Results on Special Iteration Functions

In this subsection we prove some theorems which cover many important cases. We study monotonically increasing discrete loops which are characterized by $N \in \mathbf{N}$ and the iteration functions $f_i(x)$ and we assume that $f(x) = f_{\min}(x)$ can be determined independently of x. The initial value of the loop variable is assumed to be $k_1 = 1$, but our results can easily be generalized.

Theorem 15. *If* $f(x) = \lceil \alpha x + \beta \rceil$, $\alpha > 1$, $\beta \geq 0$, *then the length of the corresponding loop sequence is bounded above by*

$$\left\lfloor \log_\alpha \left(\frac{N(\alpha - 1) + \beta}{\alpha + \beta - 1} \right) + 1 \right\rfloor .$$

Theorem 16. *If* $f(x) = \lceil \alpha x^\gamma + \beta \rceil$, $\alpha > 1$, $\beta \geq 0$, $\gamma > 1$, *then the length of the corresponding loop sequence is bounded above by*

$$\left\lfloor \log_\gamma \left((\gamma - 1) \log_\alpha N + 1 \right) + 1 \right\rfloor .$$

Theorem 17. *If* $f(x) = \lceil q(x) + \beta \rceil$, *where* $\beta \geq 0$ *and* $q(x) = \sum_i \alpha_i x^{\gamma_i}$, $\alpha_i > 1$, $\gamma_i > 1$, *then the length of the corresponding loop sequence is bounded above by*

$$\left\lfloor \log_{\gamma_m} \left((\gamma_m - 1) \log_{\alpha_m} N + 1 \right) + 1 \right\rfloor ,$$

where the index m *is defined such that* $\gamma_m = \max_i \gamma_i$.

By similar methods lower bounds for the number of iterations of monotonically increasing discrete loops can be derived.

Integrating the results of Theorems 15, 16, 17, and similar theorems into a compiler, the number of iterations of discrete loops can often be estimated at compile time, thus producing valuable information for a tool estimating the worst-case timing behavior of a real-time program and for a real-time scheduler.

2.7 Non-Monotonical Discrete Loops

Although monotonical discrete loops are interesting for their own, many applications rely on discrete loops which are not monotonical. One example is *binary search*, where the corresponding loop sequences are non-monotonical, but the number of iterations is bounded above. An Ada-like implementation using a discrete loop is shown in Figure 4. Note that we have omitted the range of the discrete loop since it does not make sense in this connection and that we have used a non-Ada-like notation for two-dimensional vectors for the loop variable `(l,u)`.

In studying binary search we will investigate how we can generalize monotonical discrete loops such that we still can guarantee that the loop completes but the corresponding loop sequence is not monotonical.

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

function binary_search(
     item: in integer;
     arr: in sort_array)
  return index is
l: index := arr'first;
u: index := arr'last;
m: index;
  -- sucessful search only
begin
  discrete (l,u) new (l,u) := (l,(l+u)/2-1) | ((l+u)/2+1,u) loop
    m := (l+u)/2;
    if item < arr(m) then
      u := m-1;
    elsif item > arr(m) then
      l := m+1;
    else
      return m;
    end if;
  end loop;
end binary_search;
```

Figure 4: Implementation of Binary Search using a Discrete Loop

2.7.1 Binary Search

The essential property of binary search is a sequence of intervals which become smaller and smaller with each iteration of the loop. The starting interval is $i_1 = [l_1, u_1] = [1, N]$ and with each iteration the interval $i_\nu = [l_\nu, u_\nu]$ is changed according to

$$i_{\nu+1} = [l_{\nu+1}, u_{\nu+1}] = \begin{cases} \left[l_\nu, \left\lfloor \frac{u_\nu + l_\nu}{2} - 1 \right\rfloor\right] \\ \left[\left\lfloor \frac{u_\nu + l_\nu}{2} + 1 \right\rfloor, u_\nu\right] \end{cases} \text{ or}$$

depending on which sub-interval contains the element being sought. If the sought element is equal to $\left\lfloor \frac{u_\nu + l_\nu}{2} \right\rfloor$, the algorithm terminates. In the worst case this is true if the interval contains just one element, i.e., if $l_\omega = u_\omega$ for some $\omega \geq 1$.

On one hand this shows a clear relationship to discrete loops, e.g. the indeterminism and the recurrently defined loop variable, on the other hand the corresponding loop sequences are non-monotonical in general. But a closer inspection shows that there is a monotonical sequence hidden in the algorithm, namely the length of the intervals.

We can even determine an upper bound for the number of loop iterations. Let $\ell_\nu = u_\nu - l_\nu + 1$ denote the length of the interval i_ν. Then

$$\ell_{\nu+1} = \max\left\{ \left\lfloor \frac{u_\nu + l_\nu}{2} \right\rfloor - l_\nu, u_\nu - \left\lfloor \frac{u_\nu + l_\nu}{2} \right\rfloor \right\}.$$

We have

$$\left\lfloor \frac{u_\nu + l_\nu}{2} \right\rfloor - l_\nu \leq \frac{u_\nu + l_\nu}{2} - l_\nu = \frac{u_\nu - l_\nu}{2}$$

and

$$u_\nu - \left\lfloor \frac{u_\nu + l_\nu}{2} \right\rfloor \leq u_\nu - \frac{u_\nu + l_\nu - 1}{2} = \frac{u_\nu - l_\nu + 1}{2}.$$

Thus

$$\ell_{\nu+1} \leq \frac{u_\nu - l_\nu + 1}{2} = \frac{\ell_\nu}{2}.$$

Mentioning $\ell_{\nu+1} \in \mathbf{N}$, we must have $\ell_{\nu+1} \leq \lfloor \ell_\nu/2 \rfloor$ and the length of the interval ℓ_ν is bounded above by U_ν which satisfies the recurrence relation

$$\begin{aligned} U_1 &= N, \\ U_{\nu+1} &= \lfloor U_\nu/2 \rfloor. \end{aligned} \tag{4}$$

Hence the number of iterations performed by binary search is bounded above by ω which is defined by $\omega = \min\{\nu : U_\nu = 0\}$.

Solving equation (4) by applying standard techniques we obtain (see e.g. [Graham et al., 1989])

$$U_\nu = \left\lfloor \frac{U_1}{2^{\nu-1}} \right\rfloor.$$

Thus $U_\nu = 0$ if $2^{\nu-1} > N$, i.e., if $\nu > \mathrm{ld}\ N + 1$. Hence the number of iterations performed by binary search is bounded above by

$$\omega = \lfloor \mathrm{ld}\ N + 2 \rfloor.$$

The ideas we have seen in studying binary search can be generalized to a new kind of discrete loop which is treated in the following section.

2.8 Discrete Loops with a Remainder Function

Definition 18. In contrast to the previous sections we now define a *loop sequence of remaining items* to be the sequence of *the number of data items* that remain to be processed during the remaining iterations of the loop. Such a loop sequence is denoted by (r_ν) and the set of all loop sequences by $\mathcal{R} = \{(r_\nu)\}$. A corresponding discrete loop is called a *discrete loop with a remainder function*.

Remark. Definition 18 is justified by the fact that normally each iteration of a loop excludes a certain number of data items from future processing (within the same loop statement). Thus the sequence of the number of the remaining items is responsible for the overall number of loop iterations. This situation is typical for *divide and conquer* algorithms. In our example of binary search the number of the remaining items is equal to the length of the remaining interval.

Definition 19. A loop sequence of remaining items is called *monotonical* if $r_{\nu+1} < r_\nu$.

Definition 20. A discrete loop with a remainder function is called *monotonical* if all its loop sequences $(r_\nu) \in \mathcal{R}$ are monotonical.

Lemma 21. *A monotonical discrete loop with a remainder function is completing.*

2.8.1 Some Examples of Monotonical Discrete Loops with Remainder Functions

The interested reader is refered to [Blieberger, 1994] for details on syntactical and semantical issues of discrete loops with remainder functions.

In the following we will give further examples of discrete loops with remainder functions.

Traversing Binary Trees

Discrete loops with remainder functions are especially well-suited for algorithms designed to traverse binary trees. A template showing such applications is given in Figure 5. In this figure `root` denotes a pointer to the root of the tree, `height`

```
discrete node_pointer := root
    new node_pointer := node_pointer.left | node_pointer.right
  with h := height
    new h := h-1 loop

  -- loop body:
  --    Here the node pointed at by node_pointer is processed
  --    and node_pointer is either set to the left or right
  --    successor.
  --    The loop is completed by an exit-statement if node_pointer = null;

end loop;
```

Figure 5: Template for Traversing Binary Trees

denotes the maximum height of the tree, and `node_pointer` is a pointer to a node of the tree. The actual value of `height` depends on which kind of tree is used, e.g. standard binary trees or AVL-trees.

Weight-Balanced Trees

Weight-balanced trees have been introduced in [Nievergelt and Reingold, 1973] and are treated in detail in [Mehlhorn, 1984b] and in [Blum and Mehlhorn, 1980].

Definition 22. We define:

1. Let T be a binary tree with left subtree T_ℓ and right subtree T_r. Then
$$\rho(T) = |T_\ell|/|T| = 1 - |T_r|/|T|$$
is called the root balance of T. Here $|T|$ denotes the number of leaves of tree T.
2. Tree T is of bounded balance α if for every subtree T' of T:
$$\alpha \leq \rho(T') \leq 1 - \alpha$$

3. BB[α] is the set of all trees of bounded balance α.

If the parameter α satisfies $1/4 < \alpha \leq 1-\sqrt{2}/2$, the operations *Access*, *Insert*, *Delete*, *Min*, and *Deletemin* take time $O(\log N)$ in BB[α]-trees. Here N is the number of leaves in the BB[α]-tree. Some of the above operations can move the root balance of some nodes on the path of search outside the permissible range $[\alpha, 1-\alpha]$. This can be "repaired" by *single* and *double rotations* (for details see [Mehlhorn, 1984b] and [Blum and Mehlhorn, 1980]).

BB[α]-trees are binary trees with bounded height. In fact it is proved in [Mehlhorn, 1984b] that

$$\text{height}(T) \leq \frac{\text{ld } N - 1}{-\text{ld } (1-\alpha)} + 1,$$

where N is the number of leaves in the BB[α]-tree T.

A template for the above operations is shown in Figure 6, where `floor(x)` is

```
discrete node_pointer := root
    new node_pointer := node_pointer.left | node_pointer.right
  with h := floor(ld(N)/(-ld(1-alpha)))+1
    new h := h-1 loop

  -- loop body

end loop;
```

Figure 6: A Template for Operations on BB[α]-trees

supposed to implement $\lfloor x \rfloor$. Since the notion of height of weight-balanced trees defined in [Mehlhorn, 1984b] is not well-suited for direct application of discrete loops, the remainder function in Figure 6 has been slightly modified.

A semantically equivalent template for traversing BB[α]-trees is shown in Figure 7. The remainder function of Figure 7 has the advantage that it does not

```
discrete node_pointer := root
    new node_pointer := node_pointer.left | node_pointer.right
  with r := N                              -- N = number of leaves of tree
    new r := floor((1-alpha)*r) loop

  -- loop body

end loop;
```

Figure 7: Another Template for Operations on BB[α]-trees

need logarithms since it works with the number of leaves instead of the height of the tree. In addition it does require less mathematical skill from the programmer.

2.9 Concluding Remarks on Discrete Loops

Since discrete loops are well-suited for determining the number of iterations, they form an ideal frame-work for estimating the worst-case timing behavior of real-time programs.

It remains to compare discrete loops with recent approaches in the domain of real-time systems. Some of these approaches have already been mentioned in the introduction.

1. Assume that the only thing that is known is $U \in \mathbf{N}$, an upper bound for the number of iterations of a general loop. Then we define the remainder function of a discrete loop by

$$r_i = U,$$
$$r_{\nu+1} = r_\nu - 1.$$

   Obviously this is semantically equivalent to the approaches described in the introduction (see [Gehani and Ramamritham, 1991, Park, 1993, Shaw, 1989, Puschner and Koza, 1989, Halang and Stoyenko, 1991]): If the upper bound U is exceeded, the exception **loop_error** is raised, which must be caught by an appropriate exception handler in order to treat this exceptional case.
2. An upper bound for the amount of time T the loop uses can be given by

$$r_i = T,$$
$$r_{\nu+1} = r_\nu - \text{time(loop_body)}, \tag{5}$$

   where time(loop_body) is the time that passed since (5) has been elaborated the last time.
   Hence the loop completes if the upper bound T has been exceeded. But an unpredictable amount of time may pass, until this fact is recognized, if the process executing the loop has been set into a waiting state by the scheduler.

Thus we have shown that discrete loops can simulate all important recent concepts that have been invented to handle general loops in the domain of real-time systems. It is, however, even more important that we have demonstrated in the previous subsections, how to use discrete loops in applications and how easy the timing behavior of discrete loops can be analyzed.

3 Real-Time Recursions

3.1 Introduction

If recursive procedures are to be used in implementing real-time applications, several problems occur:

1. It is not clear, whether a recursive procedure completes or not (see e.g. Example 5 below).
2. If it completes, it must be guaranteed that its result is delivered within a predefined deadline.

3. Since most real-time systems are embedded systems with limited storage space, the result of a recursive procedure must be computed using a limited amount of stack space.

In view of these problems most designers of real-time programming languages decide to forbid recursion in their languages, e.g. PEARL (see [DIN, 1982]), RT-Euclid (see [Kligerman and Stoyenko, 1986, Halang and Stoyenko, 1991]), Real-Time Concurrent C (see [Gehani and Ramamritham, 1991]), and the MARS-approach (see [Kopetz et al., 1989, Puschner and Koza, 1989, Vrchoticky, 1993]).

Other so-called real-time languages allow recursions to be used, but do not provide any help to the programmer in order to estimate time and space behavior of the recursive procedures, e.g. Ada (see [Ada, 1995]) and PORTAL (see [Businger, 1985]). Interestingly, a subset of Ada (see [Forsyth, 1993]) designed for determining the worst-case timing behavior forbids recursion. PORTAL uses RECURSION resources and terminates a recursive computation if the resource is exhausted. Although it is not clear from the description, one can suspect that a RECURSION resource is equivalent to an area of memory that contains the stack space. Both Ada and PORTAL cannot handle the time complexity of recursive procedures.

Other approaches do not address recursion at all (see e.g. [Mok et al., 1989, Shaw, 1989, Park, 1993, Ishikawa et al., 1990]), others propose to replace recursive algorithms by iterative ones or to transform them into non-recursive schemes by applying program transformation rules (see e.g. [Puschner and Koza, 1989]). Certainly, if a *simple* iterative version of a recursive algorithm exists and it is also superior in space and time behavior, it should be used instead of a recursive implementation. On the other hand there are the following reasons why recursive algorithms should be implemented by recursive procedures:

- The space and time behavior of transformed programs are by no means easier to investigate than their recursive counterparts, since the stack has to be simulated and because they contain while-loops. In general, the number of iterations of these loops cannot be determined at compile time, even with the use of discrete loops (see Section 2 and [Blieberger, 1994]).
- A recursive algorithm originates from recursiveness in the problem domain. From the view of software engineering, a program reflecting the problem domain is considered better than others not doing so (see e.g. [Booch, 1991]).
- Often recursive algorithms are easier to understand, to implement, to test, and to maintain than non-recursive versions.

The approach presented in [Blieberger and Lieger, 1994] is different in that recursion is not forbidden, but instead recursive procedures are constrained such that their space and time behavior either can be determined at compile time or can be checked at runtime. Thus timing errors can be found either at compile time or are shifted to logical errors detected at runtime.

The constraints mentioned above are more or less simple conditions. If they can be proved to hold, the space and time behavior of the recursive procedure can be estimated easily.

In this section we will repeat the major results of [Blieberger and Lieger, 1994] and use the following notational conventions:

- When we speak of *recursive procedures*, we mean both *recursive procedures* and *recursive functions*.

– When we speak of *space*, we mean *stack space* and not *heap space*. If dynamic data structures are used for the internal representation of an object, the space allocated from the heap is under control of the object/class manager. On the other hand, the space allocated from the stack originating from the use of recursive procedures can not be explicitly controlled by the application. This case requires a delicate treatment, which will be performed in this section.

Throughout this section we will use 4 examples to illustrate our theoretical treatment.

Example 1. The *Factorial Numbers* $n!$ given by the recursion

$$\mathrm{fac}(n) = \begin{cases} n \cdot \mathrm{fac}(n-1) & \text{if } n > 0, \\ 1 & \text{if } n = 0. \end{cases}$$

Example 2. The *Fibonacci Numbers* $f(n)$ given by the recursion $(n \geq 2)$

$$\begin{aligned} f(0) &= f(1) = 1, \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Example 3. The *Ackermann Function* $\mathcal{A}(x, y)$ given by

$$\begin{aligned} \mathcal{A}(0, y) &= y + 1 \\ \mathcal{A}(x+1, 0) &= \mathcal{A}(x, 1) \\ \mathcal{A}(x+1, y+1) &= \mathcal{A}(x, \mathcal{A}(x+1, y)) \end{aligned}$$

Example 4. A recursive version of *Mergesort*, the source code of which is shown in Figure 8. Note that the Ada source code contains a hidden for-loop, namely at line 17, and a discrete loop starting at line 18. (The syntax and semantic of discrete loops can be found in Section 2 and [Blieberger, 1994].)

Further examples will be given in the text but those listed above will be our major references.

It is obvious that the first three examples of recursive procedures introduced above will not be used in practical applications. Rather the first two will be implemented without recursion and we suppose the third one will not occur in any practical application. Nevertheless we will use these examples throughout this section because they are simple enough to illustrate our ideas. Of course this does not mean that our approach can only be applied to simple cases. In fact it is applicable to very complex and general cases as can be seen in the following subsections.

3.2 Definitions and Preliminary Results

Definition 23. Essential properties of a recursive procedure p are the *parameter space* $\mathcal{F}$, i.e., the set of all possible (tuples of) values of parameters of p, a set $\mathcal{F}_0 \subseteq \mathcal{F}$, the *terminating values* of $\mathcal{F}$, and its code. If p is called with actual parameters $f_0 \in \mathcal{F}_0$, the code being executed must not contain a recursive call of p to itself. If p is called with actual parameters $f \in \mathcal{F} \setminus \mathcal{F}_0$, the code being executed must contain at least one recursive call of p to itself.

```
1   N: constant integer := ... ;                    -- number of elements to be sorted
2   subtype index is integer range 1 .. N;
3   type gen_sort_array is array (index range <>) of ... ;
4   subtype sort_array is gen_sort_array (index);
5   sort_arr: sort_array;

6   procedure merge_sort(from,to: index) is
7     m: constant integer := (from+to)/2 + 1;
8     subtype aux_array is gen_sort_array(m..to);
9     aux: aux_array;
10    p,q,r: integer;
11  begin
12    if from = to then
13      return;
14    end if;
15    merge_sort(from,m-1);
16    merge_sort(m,to);
17    aux := sort_arr(m..to);
18    discrete (p,q,r) := (m-1,aux'last,to)
19         in reverse (m-1,aux'last,to) .. (from-1,aux'first,from)
20         new (p,q,r) := (p-1,q,r-1) | (p,q-1,r-1) loop
21      if p < from or else target(p) < aux(q) then
22         target(r) := aux(q);
23         r := r-1;
24         q := q-1;
25      else
26         target(r) := target(p);
27         r := r-1;
28         p := p-1;
29      end if;
30    end loop;
31  end merge_sort;
```

Figure 8: Ada Source Code of Mergesort using a Discrete Loop

Definition 24. We call a recursive procedure p *well-defined* if for each element of $\mathcal{F}$ the procedure p completes correctly, e.g. does not loop infinitely and does not terminate because of a runtime error (other than those predefined in this section).

From now on, when we use the term recursive procedure, we mean well-defined recursive procedure.

Definition 25. We define a set $\mathcal{R}(f) \subseteq \mathcal{F}$, $(f \in \mathcal{F} \setminus \mathcal{F}_0)$ by $\overline{f} \in \mathcal{R}(f)$ iff $p(\overline{f})$ is directly called in order to compute $p(f)$. $\mathcal{R}(f)$ is called the set of *direct successors* of f. If $f \in \mathcal{F}_0$, the set $\mathcal{R}(f) = \emptyset$, i.e., it is empty.

Remark. We assume that if $\overline{f} \in \mathcal{R}(f)$, it is not essential how often p is called with parameter $\overline{f}$. Note that it can be guaranteed by the runtime system that $p(\overline{f})$ is evaluated only once.

Definition 26. We define a sequence of sets $\mathcal{R}_k(f)$ by

$$\mathcal{R}_0(f) = \{f\}$$
$$\mathcal{R}_{k+1}(f) = \mathcal{R}_k(f) \cup \{\overline{f} \mid \overline{f} \in \mathcal{R}(g) \text{ where } g \in \mathcal{R}_k(f)\}$$

and we define the set $\mathcal{R}^*(f)$ by

$$\mathcal{R}^*(f) = \lim_{k \to \infty} \mathcal{R}_k(f).$$

We call $\mathcal{R}^*(f)$ the set of *necessary parameter values* to compute $p(f)$.

Definition 27. We define a sequence of sets $\mathcal{F}_k$ inductively by

1. $\mathcal{F}_0$ is defined as above (see Definition 23), i.e., $\mathcal{F}_0$ contains the terminating values of $\mathcal{F}$.
2. Let $\mathcal{F}_0, \ldots, \mathcal{F}_k$ be defined. Then we define $\mathcal{F}_{k+1}$ by

$$\mathcal{F}_{k+1} = \left\{ f \in \mathcal{F} \setminus \bigcup_{i=0}^{k} \mathcal{F}_i \;\middle|\; \mathcal{R}(f) \subseteq \bigcup_{i=0}^{k} \mathcal{F}_i \right\}.$$

Lemma 28. *We have* $\bigcup_{k \geq 0} \mathcal{F}_k = \mathcal{F}$.

Again, no proofs are given. They can be found in [Blieberger and Lieger, 1994].

Corollary 29. *By definition and by Lemma 28 we see that the sequence* $\mathcal{F}_k$ *partitions the set* $\mathcal{F}$, *i.e., for each* $f \in \mathcal{F}$ *holds that there exists exactly one* $k \in \boldsymbol{N}$ *such that* $f \in \mathcal{F}_k$ *and* $f \notin \mathcal{F}_i$ *for all* $i \neq k$. *Thus the* $\mathcal{F}_k$ *are* equivalence classes.

Definition 30. Let $f \in \mathcal{F}$ and let k be such that $f \in \mathcal{F}_k$, then k is called the *recursion depth* of $p(f)$. We write $k = \text{recdep}(f)$. For $f, g \in \mathcal{F}$, we write $f \approx g$ iff $\text{recdep}(f) = \text{recdep}(g)$.

Definition 31. A recursive procedure p is called *monotonical* if for all $f_k \in \mathcal{F}_k$ and for $f_i \in \mathcal{F}_i$, $0 \leq i < k$, we have $f_i \prec f_k$, where "$\prec$" is a suitable binary relation that satisfies for all $f_1, f_2, f_3 \in \mathcal{F}$

1. either $f_1 \prec f_2$ or $f_2 \prec f_1$ or $f_1 \approx f_2$ and
2. if $f_1 \prec f_2$ and $f_2 \prec f_3$, then $f_1 \prec f_3$.

We write $f_1 \preceq f_2$ if either $f_1 \prec f_2$ or $f_1 \approx f_2$.

Remark. If p is a monotonical recursive procedure, then $\overline{f} \prec f$ for all $\overline{f} \in \mathcal{R}(f)$.

Example 1. For the Factorial Numbers we have $\mathcal{F} = \mathbf{N}$, $\mathcal{R}(k) = \{k-1\}$, and $\mathcal{F}_0 = \{0\}$, $\mathcal{F}_k = \{k\}$. Furthermore we have $\text{recdep}(k) = k$ and the "$\prec$"-relation for $\mathcal{F}$ is the "$<$"-relation for integers.

Example 2. For the Fibonacci Numbers we obtain $\mathcal{F} = \mathbf{N}$, $\mathcal{R}(k) = \{k-1, k-2\}$, and $\mathcal{F}_0 = \{0, 1\}$, $\mathcal{F}_k = \{k+1\}$. Furthermore we have $\text{recdep}(k) = k-1$, if $k \geq 1$, the "$\prec$"-relation for $\mathcal{F}$ is the "$<$"-relation for integers.

Example 3. The Ackermann Function gives

$$\mathcal{F} = \mathbf{N}^2,$$
$$\mathcal{R}((x,y)) = \begin{cases} \emptyset & \text{if } x = 0, \\ \{(x-1, \mathcal{A}(x,y-1)), (x,y-1)\} & \text{if } y \geq 1, \\ \{(x-1,1)\} & \text{if } y = 0 \end{cases}$$

and (see [Lieger and Blieberger, 1994], where proofs of the following facts can be found)

$$\mathcal{F}_0 = \{(0,y) \mid y \in \mathbf{N}\},$$
$$\mathcal{F}_k = \{(x,y) \mid \mathcal{A}(x,y) + x - 2 = k, x > 0\}.$$

Furthermore we have

$$\text{recdep}((0,y)) = 0 \text{ and for } x > 0$$
$$\text{recdep}((x,y)) = \mathcal{A}(x,y) + x - 2,$$

the "$\prec$"-relation for $\mathcal{F}$ is

$$(x_1,y_1) \prec (x_2,y_2) \Leftrightarrow \mathcal{A}(x_1,y_1) + x_1 < \mathcal{A}(x_2,y_2) + x_2$$

if $x_1, x_2 > 0$.

Example 4. For Mergesort we derive

$$\mathcal{F} = \mathbf{N}^2,$$
$$\mathcal{R}((x,y)) = \left\{ \left(x, \left\lfloor \frac{x+y}{2} \right\rfloor \right), \left(\left\lfloor \frac{x+y}{2} \right\rfloor + 1, y \right) \right\},$$

and

$$\mathcal{F}_0 = \{(x,x) \mid x \in \mathbf{N}\},$$
$$\mathcal{F}_k = \{(x,y) \mid 2^{k-1} \leq (y-x) < 2^k\}.$$

Furthermore we have

$$\text{recdep}((x,y)) = \lceil \text{ld } (y-x+1) \rceil,$$

the "$\prec$"-relation for $\mathcal{F}$ is given by

$$(x_1,y_1) \prec (x_2,y_2) \Leftrightarrow y_1 - x_1 < y_2 - x_2,$$

where "$<$" denotes the "$<$"-relation of integer numbers.

Example 5. An interesting example is the "wondrous" (see [Hofstadter, 1979]) function. This function is not known to be well-defined, but we will study it anyway since it has interesting properties. It is defined by

$$d(n) = \begin{cases} d(n/2) & \text{if } n \equiv 0(2) \text{ and} \\ d(3n+1) & \text{if } n \equiv 1(2). \end{cases}$$

It has been conjectured that finally the "wondrous" function finds itself repeating the three numbers 4, 2, and 1 infinitely, irrespective of the initial value $n \in \mathbf{N}$. This, however, has not been proved.

Now defining a (possibly not well-defined) recursive procedure by

$$\begin{aligned} \mathcal{F} &= \mathbf{N}, \\ \mathcal{F}_0 &= \{1,2,4\}, \text{ and} \\ \mathcal{R}(k) &= \begin{cases} k/2 & \text{if } k \equiv 0(2) \\ 3k+1 & \text{if } k \equiv 1(2), \end{cases} \end{aligned}$$

we obtain

$$\begin{aligned} &\mathcal{F}_1 = \{8\},\ \mathcal{F}_2 = \{16\},\ \mathcal{F}_3 = \{5,32\}, \\ &\mathcal{F}_4 = \{10,64\},\ \mathcal{F}_5 = \{3,20,21,128\}, \ldots\ . \end{aligned}$$

It is not obvious how recdep(n) and a suitable "$\prec$"-relation can be expressed by a simple formula.

3.3 Computational Model and Space and Time Effort

The time effort $\mathcal{T}$ of a recursive procedure p is a recursive function

$$\mathcal{T} : \mathcal{F} \to \mathbf{R}$$

or

$$\mathcal{T} : \mathcal{F} \to \mathbf{N}.$$

If time is measured in integer multiples of say micro-seconds or CPU clock ticks, one can use an integer valued function $\mathcal{T}$ instead of a real valued one.

In a similar way $\mathcal{S}$, the space effort of p, is a recursive function

$$\mathcal{S} : \mathcal{F} \to \mathbf{N},$$

where space is measured in multiples of bits or bytes.

Both functions $\mathcal{T}$ and $\mathcal{S}$ are defined recursively depending on the source code of p. How the recurrence relations for $\mathcal{T}$ and $\mathcal{S}$ are derived from the source code and which statements are allowed in the source code of p, is described in the following subsection.

3.3.1 Recurrence Relations for $\mathcal{S}$ and $\mathcal{T}$

The source code of a recursive procedure is considered to consist of

- simple segments of linear code, the performance of which is known a priori,
- if-statements,
- loops with known upper bounds of the number of iterations which can be derived at compile time, e.g. for-loops or discrete loops (see Section 2 and [Blieberger, 1994]), and
- recursive calls to the procedure itself.

In terms of a context-free grammar this is stated as follows

$$
\begin{aligned}
\mathit{code}(f) &::= \textbf{if } f \in \mathcal{F}_0 \textbf{ then } \mathit{nonrecursive}(f) \textbf{ else } \mathit{recursive}(f) \textbf{ end if} \\
\mathit{recursive}(f) &::= \mathit{seq}(f) \\
\mathit{seq}(f) &::= \mathit{statement}(f) \ \{\mathit{statement}(f)\} \\
\mathit{statement}(f) &::= \mathit{simple}(f) \mid \mathit{compound}(f) \mid \mathit{rproc}(f \to \overline{f}) \\
\mathit{compound}(f) &::= \mathit{ifs}(f) \mid \mathit{bloops}(f) \\
\mathit{ifs}(f) &::= \textbf{if } \mathit{cond}(f) \textbf{ then } \mathit{seq}(f) \textbf{ else } \mathit{seq}(f) \textbf{ end if} \\
\mathit{bloops}(f) &::= \textbf{loop } <\mathit{bound}(f)> \mathit{seq}(f)
\end{aligned}
$$

The syntax of $\mathit{nonrecursive}(f)$ is defined exactly the same way but $\mathit{rproc}(f \to \overline{f})$ is not part of $\mathit{statement}(f)$. By $f \to \overline{f}$ we denote that the parameters $\overline{f}$ are used for the recursive call.

We use these definitions to derive a recurrence relation for the time effort $\mathcal{T}$:

$$\mathcal{T}(f) = \tau[f \in \mathcal{F}_0] + \tau[\mathit{nonrecursive}(f)] \quad \text{if } f \in \mathcal{F}_0,$$

where the first τ-constant comes from the evaluating the condition whether f belongs to the terminating values or not and is known a priori; the second one can be computed using the method described below, but without giving rise to a recurrence relation,

$$\mathcal{T}(f) = \tau[f \in \mathcal{F}_0] + \tau[\mathit{recursive}(f)] \quad \text{if } f \notin \mathcal{F}_0,$$

where

$$
\begin{aligned}
&\mathcal{T}[\mathit{recursive}(f)] = \mathcal{T}[\mathit{seq}(f)] \\
&\mathcal{T}[\mathit{seq}(f)] = \sum \mathcal{T}[\mathit{statement}(f)] \\
&\mathcal{T}[\mathit{ifs}(f)] = \mathcal{T}[\mathit{cond}(f)] + \begin{cases} \mathcal{T}[\mathit{seq}_{\texttt{True}}(f)] & \text{if the condition evaluates to true,} \\ \mathcal{T}[\mathit{seq}_{\texttt{False}}(f)] & \text{otherwise.} \end{cases} \\
&\mathcal{T}[\mathit{bloops}(f)] = <\mathit{bound}(f)> \cdot \mathcal{T}[\mathit{seq}(f)] \\
&\mathcal{T}[\mathit{simple}(f)] = \tau(\mathit{simple}) \\
&\mathcal{T}[\mathit{rproc}(f \to \overline{f})] = \mathcal{T}(\overline{f})
\end{aligned}
$$

where $\tau(\mathit{simple})$ is known a priori.

Note that $<\mathit{bound}(f)>$ may depend on f, e.g. a for-loop with iterations depending on f.

The recurrence relation for the space effort $\mathcal{S}$ is given by:

$$\mathcal{S}(f) = \mathcal{S}(decl_part(f)) + \max(\sigma[f \in \mathcal{F}_0], \sigma[nonrecursive(f)]) \quad \text{if } f \in \mathcal{F}_0,$$

where the first σ-constant is known a priori and the second one can be computed in a similar way as shown below, but without giving rise to a recurrence relation,

$$\mathcal{S}(f) = \mathcal{S}(decl_part(f)) + \max(\sigma[f \in \mathcal{F}_0], \sigma[recursive(f)]) \quad \text{if } f \notin \mathcal{F}_0,$$

where

$$\mathcal{S}[recursive(f)] = \mathcal{S}[seq(f)]$$
$$\mathcal{S}[seq(f)] = \max(\mathcal{S}[statement(f)])$$
$$\mathcal{S}[ifs(f)] = \begin{cases} \max(\mathcal{S}[cond(f)], \mathcal{S}[seq_{\texttt{True}}(f)]) & \text{if the condition evaluates to true,} \\ \max(\mathcal{S}[cond(f)], \mathcal{S}[seq_{\texttt{False}}(f)]) & \text{otherwise.} \end{cases}$$
$$\mathcal{S}[bloops(f)] = \max(\mathcal{S}[seq(f)])$$
$$\mathcal{S}[simple(f)] = \sigma(simple)$$
$$\mathcal{S}[rproc(f \to \overline{f})] = \mathcal{S}(\overline{f})$$

where $\sigma(simple)$ is known a priori and $\mathcal{S}(decl_part(f))$ denotes the space effort of the declarative part of the recursive function, e.g. space used by locally declared variables. Note that the space effort of the declarative part may depend on f, since one can declare arrays of a size depending on f for example.

3.3.2 Monotonical Space and Time Effort

Given some actual parameters $f \in \mathcal{F}$, $\mathcal{T}(f)$ and $\mathcal{S}(f)$ can easily be determined at compile time. This can even be done if only upper and lower bounds of f exist, e.g. $l \preceq f \preceq u$, $l, u \in \mathcal{F}$, since $\max_{l \preceq f \preceq u} \mathcal{T}(f)$ and $\max_{l \preceq f \preceq u} \mathcal{S}(f)$ can be computed effectively.

Definition 32. If $f_1 \preceq f_2$ implies $\mathcal{S}(f_1) \leq \mathcal{S}(f_2)$ and $\mathcal{T}(f_1) \leq \mathcal{T}(f_2)$, we call the underlying recursive procedure *globally space-monotonical* and *globally time-monotonical*, respectively.

Corollary 33. *Note that $f_1 \approx f_2$ implies $\mathcal{S}(f_1) = \mathcal{S}(f_2)$ and $\mathcal{T}(f_1) = \mathcal{T}(f_2)$, respectively.*

There are two cases:

1. $\mathcal{S}$ and $\mathcal{T}$ can be shown to be monotonical at compile-time and
2. $\mathcal{S}$ and $\mathcal{T}$ can be solved at compile-time and the (non-recursive) solution can be proved to be monotonical.

In both cases we clearly have:

Theorem 34. *If p is globally space or time-monotonical, then*

$$\overline{\boldsymbol{S}}(l,u) = \max_{l \preceq f \preceq u} \mathcal{S}(f) = \max_{g \approx u} \mathcal{S}(g)$$

and

$$\overline{\boldsymbol{T}}(l,u) = \max_{l \preceq f \preceq u} \mathcal{T}(f) = \max_{g \approx u} \mathcal{T}(g),$$

respectively.

The difference between case (1) and (2) is that in case (2) Theorem 34 can even be applied during runtime, e.g., when generic objects are instantiated (see e.g. [Ada, 1995, Ellis and Stroustrup, 1990]), while in case (1) for real-time applications Theorem 34 can only be applied at compile time, because case (1) requires one or more recursive evaluations of $\mathcal{S}$ or $\mathcal{T}$.

If no proofs are available at compile time that p is globally space or time-monotonical, runtime tests can be performed. Of course this requires some overhead for computing the result of a recursive call to p.

In the following sections we will define "local" conditions. If these conditions hold, the underlying recursive procedure is called *locally space* or *locally time-monotonical*. It will turn out that if a recursive procedure is locally space (time) monotonical, then it is also globally space (time) monotonical. (It is worth noting that the converse is not true, i.e., if a certain recursive procedure is globally space or time monotonical, it need not be locally space or time monotonical.)

Thus it suffices to prove that a certain recursive procedure is locally space or time-monotonical, before Theorem 34 can be applied. This proof often is simpler than proving the corresponding global property.

If the local properties can be proved at compile time, Theorem 34 can be applied at compile time. If there is a (non-recursive) solution of $\mathcal{S}$ or $\mathcal{T}$ known and verified at compile time, Theorem 34 can also be applied at runtime.

In addition, the local properties can be checked at runtime, such that it is not necessary to have proofs at compile time. Rather an appropriate exception is raised at runtime when the runtime system finds that the local property does not hold in a particular case. Thus timing errors are shifted to runtime errors or in other words timing errors become testable.

The major advantages of local properties are that

- they can easily be proved at compile time and
- they are well-suited for real-time applications.

In the following sections we give several examples of how easy these proofs can be derived. We think that in many cases they can be found by a (smart) compiler. In general, proofs of global properties and solving recurrence relations are more difficult.

3.4 The Space Effort of Recursive Procedures

Definition 35. Let p be a recursive procedure. We define the function $\mathcal{D} : \mathcal{F} \to \mathbf{N}$ such that $\mathcal{D}(f)$ denotes the space being part of the declarative part of p if p is called with parameter f.

The general form of $\mathcal{S}(f)$ simplifies to

$$\mathcal{S}(f) = \sigma_0' \quad \text{if } f \in \mathcal{F}_0$$
$$\mathcal{S}(f) = \mathcal{D}(f) + \max\left(\sigma_{\max}, \mathcal{S}(\overline{f}_1), \ldots, \mathcal{S}(\overline{f}_m)\right) \quad \text{if } f \notin \mathcal{F}_0,$$

where $\mathcal{R}(f) = \{\overline{f}_1, \ldots, \overline{f}_m\}$. Since the $\sigma_{\max}$-term is present in all $\mathcal{S}(f)$ provided that $f \notin \mathcal{F}_0$, we obtain

$$\mathcal{S}(f) = \sigma_0 \quad \text{if } f \in \mathcal{F}_0 \tag{6}$$
$$\mathcal{S}(f) = \mathcal{D}(f) + \max\left(\mathcal{S}(\overline{f}_1), \ldots, \mathcal{S}(\overline{f}_m)\right) \quad \text{if } f \notin \mathcal{F}_0, \tag{7}$$

where $\sigma_0 = \max(\sigma_0', \sigma_{\max})$. Note that this does not change the value of $\mathcal{S}(f)$ if $f \in \mathcal{F} \setminus \mathcal{F}_0$.

Remark. Evaluating $\mathcal{S}(f)$ for recursive functions increases the height of the stack if the recursive call is part of an expression, but this can be avoided by introducing temporary variables in the declarative part of the recursive function. (Note that this can be done at compile time!)

Definition 36. For each $f \in \mathcal{F}$ the *recursion digraph* $\mathcal{G}(f)$ is defined by the set of vertices $V = \mathcal{R}^*(f)$ and the set of edges $E = \{(g, \overline{g}) \mid g, \overline{g} \in V \text{ and } \overline{g} \in \mathcal{R}(g)\}$. Each vertex g is weighted by $\mathcal{D}(g)$.

Remark. Let $\mathcal{M}$ denote the path from f to some $f_0 \in \mathcal{F}_0$, $f_0 \in \mathcal{R}^*(f)$ with maximum weight $W(f) = \sum_g \mathcal{D}(g)$, where g runs through all vertices on $\mathcal{M}$. Then $W(f)$ is equal to $\mathcal{S}(f)$.

Remark. Using $\mathcal{G}(f)$, the quantity $\mathcal{S}(f)$ can be computed off-line at compile time in $O(\|V\| + \|E\|)$ time (see e.g. [Mehlhorn, 1984a]).

Definition 37. Let p be a monotonical recursive procedure. We define $\mathcal{N} : \mathcal{F} \to \mathcal{F}$ to be a function such that $\mathcal{N}(f) = f_{\max}$, where $f_{\max}$ is such that $\mathrm{recdep}(f_{\max}) = \mathrm{recdep}(f) - 1$, $\mathcal{D}(f_{\max}) = \max_{\overline{f} \in \mathcal{R}(f)} \mathcal{D}(\overline{f})$, and $\mathrm{recdep}(\overline{f}) = \mathrm{recdep}(f) - 1$ in the max-term.

Definition 38. We call a monotonical recursive procedure p *locally space-monotonical* if $f_1 \prec f_2$ implies $\mathcal{D}(f_1) \le \mathcal{D}(f_2)$ and, if $f_1 \approx f_2$ and $\mathcal{D}(f_1) \le \mathcal{D}(f_2)$ implies $\mathcal{D}(\mathcal{N}(f_1)) \le \mathcal{D}(\mathcal{N}(f_2))$.

Corollary 39. *If $\mathcal{D}(f)$ is constant, then the underlying recursive procedure is locally space-monotonical.*

Theorem 40. *If p is a locally space-monotonical recursive procedure, then*

$$\mathcal{S}(f) = \sigma_0 + \sum_{0 \le k < recdep(f)} \mathcal{D}(\mathcal{N}^{(k)}(f)),$$

where $\mathcal{N}^{(k)}$ is the kth iterate of $\mathcal{N}$ and for simplicity $\mathcal{N}^{(0)}(f) = f$.

Corollary 41. *If $\mathcal{D}(f) = d$, $d \in \boldsymbol{N}$ constant for all $f \in \mathcal{F}$,*

$$\mathcal{S}(f) = \sigma_0 + d \cdot \mathit{recdep}(f).$$

Remark. Theorem 40 and Corollary 41 show the intuitively clear connection between the recursion depth (the height of the stack) and the space complexity of recursive procedures.

Theorem 42. *If p is locally space-monotonical, then*

$$\overline{\boldsymbol{S}}(l,u) = \max_{l \preceq f \preceq u} \mathcal{S}(f) = \max_{g \approx u} \mathcal{S}(g).$$

In the following examples the constants σ_0, σ_d, and $\tilde{\sigma}$ are derived from the (source) code of the recursive procedures.

Example 1. For the Factorial Numbers we get $\mathcal{D}(n) = \sigma_d$, constant. Thus they are locally space-monotonical (see Corollary 39) and we can even show that

$$\begin{aligned} \mathcal{S}(0) &= \sigma_0, \\ \mathcal{S}(n) &= \sigma_d + \mathcal{S}(n-1). \end{aligned}$$

Mentioning $\mathrm{recdep}(n) = n$ and $\mathcal{N}(n) = n - 1$ we derive

$$\mathcal{S}(n) = \sigma_0 + \sum_{k=0}^{n-1} \sigma_d = \sigma_0 + n \cdot \sigma_d.$$

Example 2. For the Fibonacci Numbers we obtain $\mathcal{D}(n) = \sigma_d$, constant. Thus they are locally space-monotonical (see Corollary 39) and we can even show that

$$\begin{aligned} \mathcal{S}(0) &= \mathcal{S}(1) = \sigma_0, \\ \mathcal{S}(n) &= \sigma_d + \mathcal{S}(n-1). \end{aligned}$$

Mentioning $\mathrm{recdep}(n) = n - 1$ and $\mathcal{N}(n) = n - 1$ we derive for $n \geq 1$

$$\mathcal{S}(n) = \sigma_0 + \sum_{k=0}^{n-2} \sigma_d = \sigma_0 + (n-1) \cdot \sigma_d.$$

Example 3. Since $\mathcal{D}((x,y)) = \sigma_d$, constant, the Ackermann Function is locally space-monotonical (see Corollary 39). In addition, since

$$\mathcal{N}((x,y)) = (x-1, \mathcal{A}(x, y-1)),$$

we get for the Ackermann Function (see [Lieger and Blieberger, 1994])

$$\begin{aligned} \mathcal{S}((0,y)) &= \sigma_0, \\ \mathcal{S}((x,y)) &= \sigma_0 + \sigma_d \cdot (\mathcal{A}(x,y) + x - 2). \end{aligned}$$

Example 4. Mergesort is treated a little inexactly. However an exact treatment is possible by use of parameter space morphisms which are introduced in [Blieberger and Lieger, 1994].

Writing $n = y - x + 1$ we get $\mathcal{D}(n) = \sigma_d + \lfloor n/2 \rfloor \tilde{\sigma}$. Thus Mergesort is locally space-monotonical.

But we can also determine the exact behavior of Mergesort. We obtain

$$\begin{aligned}\mathcal{S}((x,x)) &= \sigma_0\,,\\ \mathcal{S}((x,y)) &= \sigma_d + \left(y - \left\lceil \frac{x+y}{2} \right\rceil\right)\tilde{\sigma} + \mathcal{S}\left(\left(x, \left\lfloor \frac{x+y}{2} \right\rfloor\right)\right).\end{aligned}$$

because

$$\mathcal{N}((x,y)) = (x, \lceil (x+y)/2 \rceil).$$

Since $\mathcal{S}(x,y)$ does only depend on the length of the array under consideration, we write again $n = y - x + 1$ and obtain

$$\begin{aligned}\mathcal{S}(1) &= \sigma_0\,,\\ \mathcal{S}(n) &= \sigma_d + \lfloor n/2 \rfloor \tilde{\sigma} + \mathcal{S}(\lceil n/2 \rceil).\end{aligned}$$

This can be solved and we finally get

$$\mathcal{S}(n) = \sigma_0 + \lceil \mathrm{ld}\ n \rceil \sigma_d + (n-1)\tilde{\sigma}.$$

3.5 The Time Effort of Recursive Procedures

Denoting by $\tau(f)$, $f \in \mathcal{F}$ the time used to perform $p(f)$ without taking into account the recursive calls, we have

$$\mathcal{T}(f) = \tau(f) + \sum_{\overline{f} \in \mathcal{R}(f)} \mathcal{T}(\overline{f}).$$

Definition 43. For all $f_1, f_2 \in \mathcal{F}$ we write $f_1 \sqsubseteq f_2$ (or equivalently $f_2 \sqsupseteq f_1$) if $f_1 \preceq f_2$ and $\tau(f_1) \le \tau(f_2)$.

Definition 44. Let $f_1, f_2 \in \mathcal{F}$, $\mathcal{R}(f_i) = \{f_{i,1}, \ldots, f_{i,m_i}\}$, $i = 1,2$, such that $f_{i,1} \sqsupseteq f_{i,2} \sqsupseteq \ldots \sqsupseteq f_{i,m_i-1} \sqsupseteq f_{i,m_i}$, $i = 1,2$.

If for all $f_1 \sqsubseteq f_2$, we have $m_1 \le m_2$ and $f_{1,r} \sqsubseteq f_{2,r}$, $r = 1, \ldots, m_1$, then the underlying recursive procedure is called *locally time-monotonical.*

Corollary 45. *If for all $f_1, f_2 \in \mathcal{F}$ $f_1 \prec f_2$ implies $\tau(f_1) \le \tau(f_2)$ and if $\|\mathcal{R}(f)\| \le 1$ for all $f \in \mathcal{F}$, then the underlying recursive procedure is locally time-monotonical.*

Lemma 46. *If a monotonical recursive procedure p is locally time-monotonical, then $f_1 \sqsubseteq f_2$ implies $\mathcal{T}(f_1) \le \mathcal{T}(f_2)$.*

Corollary 47. *If we have $f_1 \sqsubseteq f_2$ and $f_2 \sqsubseteq f_1$, we conclude that $f_1 \approx f_2$ and $\tau(f_1) = \tau(f_2)$. By Lemma 46 this implies $\mathcal{T}(f_1) = \mathcal{T}(f_2)$.*

Lemma 46 enables us to find upper and lower bounds of the timing behavior if a range of parameter values is given.

Theorem 48. *If p is locally time-monotonical, then*

$$\overline{\boldsymbol{T}}(l,u) = \max_{l \preceq f \preceq u} \mathcal{T}(f) = \max_{g \approx u} \mathcal{T}(g).$$

In the following examples the constants τ_0, τ_1, τ_2, and τ_d are derived from the (source) code of the recursive procedures.

Example 1. Because of Corollary 45 the Factorial Numbers are locally time-monotonical.

In addition, we get

$$\begin{aligned} \mathcal{T}(0) &= \tau_0, \\ \mathcal{T}(n) &= \tau_d + \mathcal{T}(n-1). \end{aligned}$$

Mentioning $\mathrm{recdep}(n) = n$ we derive

$$\mathcal{T}(n) = \tau_0 + \sum_{k=0}^{n-1} \tau_d = \tau_0 + n \cdot \tau_d.$$

Example 2. It is easy to see that the Fibonacci Numbers are locally time-monotonical.

In addition, we derive

$$\begin{aligned} \mathcal{T}(0) &= \mathcal{T}(1) = \tau_0, \\ \mathcal{T}(n) &= \tau_d + \mathcal{T}(n-1) + \mathcal{T}(n-2). \end{aligned}$$

Thus for $n \geq 2$

$$\mathcal{T}(n) = f(n)\tau_0 + (f(n) - 1)\tau_d,$$

where $f(n)$ denotes the nth Fibonacci Number.

Example 3. It turns out that the Ackermann Function is *not* locally and *not* globally time-monotonical (see [Lieger and Blieberger, 1994]). The following gives a simple counter-example:

Let $(x_1, y_1) = (1, 13)$ and $(x_2, y_2) = (3, 1)$. Because of $\mathrm{recdep}((1,13)) = 14 = \mathrm{recdep}((3,1))$ and (for all reasonable implementations) $\tau(1,13) = \tau(3,1)$ we find that $(1,13) \approx (3,1)$ and $(1,13) \sqsubseteq (3,1)$ as well as (for reasons of symmetry) $(1,13) \sqsupseteq (3,1)$ (see Corollary 47).

Now $\mathcal{R}((1,13)) = \{(1,12),(0,14)\}$ and $\mathcal{R}((3,1)) = \{(3,0),(2,5)\}$.

As expected $\mathrm{recdep}(1,12) = \mathrm{recdep}(2,5) = 13 = 14 - 1$, $\tau(1,12) = \tau(2,5)$ and therefore $(1,12) \sqsubseteq (2,5)$ and $(1,12) \sqsupseteq (2,5)$.

Unfortunately $\mathrm{recdep}((0,14)) = 1 \neq 6 = \mathrm{recdep}((3,0))$ and thus $(0,14) \not\sqsupseteq (3,0)$, which contradicts Corollary 47 and Corollary 33.

Example 4. Writing $n = y - x + 1$, we have $\tau(n) = \tau_1 + n\tau_2$. Clearly, if $n_1 < n_2$, then $\tau(n_1) < \tau(n_2)$. This together with the fact that the length of the subarrays is $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ shows that Mergesort is locally time-monotonical.

In addition, we are able to show that

$$\mathcal{T}(1) = \tau_0$$
$$\mathcal{T}(n) \leq \tau_1 + n\tau_2 + \mathcal{T}(\lfloor n/2 \rfloor) + \mathcal{T}(\lceil n/2 \rceil).$$

The "$\leq$" originates from the fact that we can only find an upper bound for the number of iterations of the discrete loop from line 18 to 30 in Figure 8. The above recurrence relation can be solved and we finally get

$$\mathcal{T}(n) \leq n\tau_0 + (n-1)\tau_1 + \left(n - 2^{\lceil \mathrm{ld}\ n \rceil} + n\lceil \mathrm{ld}\ n \rceil\right)\tau_2.$$

3.6 Parameter Space Morphisms

The theoretical results of the previous subsections are impressive in that they are valid for recursive procedures with very general parameter space. For many applications, however, only a small "part" of the parameter space is responsible for the space and time behavior of the recursive procedure. It is possible to "abstract" from unnecessary details of the parameter space. Details can be found in [Blieberger and Lieger, 1994].

3.7 Concluding Remarks on Real-Time Recursion

Note that Theorems 42 and 48 are valid although we do *not* study *static* bounds of space and time behavior. This contrasts [Puschner and Koza, 1989], where the execution time of code blocks is estimated statically without taking into account that the execution time may depend on certain parameters (or global data). Anyway, the MARS approach [Puschner and Koza, 1989, Vrchoticky, 1993] excludes recursions.

In [Park, 1993] such information on data influencing execution time can be incorporated into the program by means of program path analysis, but [Park, 1993] does not address recursion at all.

4 Specifying Worst-Case Performance Estimates on the Level of Classes

4.1 Introduction

Generally speaking, the interest of this section is in object-oriented real-time programs. To be more specific, we are interested in how *timing information* provided at an object's interface can be used to estimate the overall timing behavior of real-time programs and most important how this timing information should be presented.

Besides we touch the problem of estimating the stack space behavior of object-oriented programs, which is of special interest for embedded systems with (of course) limited storage space.

Object-oriented real-time systems have until now only been studied in a very restricted manner. For example, in [Gwinn, 1992] the pros and cons of object-oriented real-time systems have been discussed. Summarizing this paper, there is no obvious reason why object-oriented features should not be used in real-time applications.

In [Maclean and Smith, 1995] the advantages of introducing object-oriented programming into embedded systems have been presented. In particular the authors have pointed out that object-oriented programming results in fewer programming errors, enhanced portability, and improved reuse. Since most embedded systems are classified real-time, this holds for real-time systems too.

In this section we summarize the major results of [Blieberger, 1995] in order to show that objects and classes supplied with appropriate timing information are an excellent means to analyze the timing behavior of real-time programs.

4.2 Our Object Model

We think that the concepts of *active objects* and *real-time objects* are completely orthogonal. Most real-time programming languages, such as Real-Time C++ ([Ishikawa et al., 1990]), only support *active real-time objects*. Other so-called object-oriented real-time programming languages, like Ada ([Ada, 1995]), do not even support *real-time objects*.

Our proposed orthogonalism implies the existence of *passive real-time objects*, which may be thought of as being non-active objects (no tasks or processes) that have a known worst-case timing behavior. One major advantage of passive real-time objects is that they are excellent candidates for *reuse*. Active real-time objects are less likely for this purpose because usually they are tailored to fit more precisely the problem domain than passive real-time objects. Examples for passive real-time objects include *balanced tree-structures*, *complex data structures (lists, ...)*, *real-time databases*, and so on.

Obviously there is another non-negligible issue for real-time systems, *schedulability*, which is equally important for both active and passive real-time objects. There are many excellent results on scheduling theory (see e.g. [Mok, 1984, Liu and Layland, 1973, Stoyenko and Marlowe, 1992, Xu and Parnas, 1993]). In this section, however, we are not concerned with scheduling. Our interest focuses on real-time objects, especially on passive real-time objects.

In order to study timing analysis of real-time objects, we discriminate between two views of objects:

1. the view from inside and
2. the view from outside.

Concerning (1) the *worst-case performance (WCP)* of the object is estimated with help of a WCP-tool which facilitates the timing analysis. For example compare [Halang and Stoyenko, 1991]. Often the task of the tool is eased by restricting programming language features. For example by forbidding while-loops and recursion. In order to ease the task of the WCP-tool further, it can use information provided at the interface of other objects/classes which it encounters during analyzing the code. This information provided at the interface of objects/classes forms the outside view (2) of objects.

Clearly, the WCP-tool mentioned above also has to check whether the information specified at the object's interface conforms to the values derived from

its implementation. In fact both timing and space estimates of the specification must always be greater than or equal to the values of the implementation, otherwise no executable program should be generated.

Separating specification and implementation of timing information has several advantages:

- It improves modularization and facilitates testing of the real-time system.
- During system design the specification part can be used to express timing constraints implied by the process. During implementation the compiler checks whether these constraints are met.

In the following subsections we are concerned with how this additional information of the outside view is provided.

4.3 Timing Information

In this subsection we study how the timing information of an object looks like. In particular, which "building blocks" form such a timing information, from where these blocks originate, and which kind of type these blocks can be supposed to be.

Clearly we can identify two sources from which timing information of an object originate:

1. performance characteristics of the underlying CPU and
2. performance characteristics of the algorithms and data structures used for implementing the objects behavior.

Since reuse is very important, it is clear that these two performance characteristics should be separated as much as possible.

For machine dependent performance characteristics this means that objects need a specification of timing information for each CPU they are supposed to support. How machine-dependent performance characteristics can be handled in an object-oriented framework, is treated in [Blieberger, 1995]. In addition, issues of inheritance and WCP estimates are discussed in [Blieberger, 1995].

Performance characteristics of algorithms and data structures have been studied extensively. For example see [Knuth, 1973a, Knuth, 1981, Knuth, 1973b, Mehlhorn, 1984b, Vitter and Flajolet, 1990, Aho et al., 1974, Aho et al., 1983, Sedgewick, 1988]. Commonly, this analysis is performed by informally introducing some sort of *complexity measure* (see [Vitter and Flajolet, 1990]) or *size* (see [Mehlhorn, 1984b, Aho et al., 1974]) of the data structure.

Summarizing our research on loops (see Section 2 and [Blieberger, 1994]) and recursions (see Section 3 and [Blieberger and Lieger, 1994]) in the real-time domain, we are led to the following conjecture.

Proposition 49. *The relevant "parameters" of algorithms and data structures to analyze stack space and timing behavior can be expressed in terms of* natural numbers.

Foundation. There are two problematic issues in determining the timing behavior of algorithms with respect to real-time applications: *loops* and *recursions.*

- Clearly, the number of iterations of a loop statement can be expressed by a natural number.
- For real-time recursive procedures similar statements are true. For a more detailed discussion see [Blieberger and Lieger, 1994].

4.4 WCP Estimates

Before we go into the details of this subsection, we give some useful definitions.

4.4.1 Definitions and Theoretical Results

Definition 50. Given two points of time, t_1 and t_2, $t_1 \leq t_2$, we call $\Delta = t_2 - t_1$ a *duration*. By $\mathcal{D}$ we denote the set of all possible durations.

Definition 51. A certain object $\mathcal{O}$ consists of a number of methods $\mathcal{M}_i$, $i = 1, \ldots, e$, providing its *behavior*, and its *internal state* $\mathcal{S} = \mathcal{S}(\mathcal{O})$. The implementation details of $\mathcal{M}_i$ are denoted by $\mathcal{I}(\mathcal{M}_i)$.

Definition 52. A *Worst-Case Performance* estimate for a certain method $\mathcal{M}_i$ of object $\mathcal{O}$ is a function from the set of tuples of parameters of the method ($\mathcal{F}_i$), the current internal state of $\mathcal{O}$ and the implementation details of $\mathcal{M}_i$ to $\mathcal{D}$

$$\mathsf{WCP} : \mathcal{F}_i \times \mathcal{S}(\mathcal{O}) \times \mathcal{I}(\mathcal{M}_i) \rightarrow \mathcal{D}.$$

Remark. Obviously, if WCP does not depend on $\mathcal{F}_i$ and on $\mathcal{S}(\mathcal{O})$ and the implementation is straight-forward, the estimate reduces to a (machine-dependent) constant. Usually, only this simple case is studied.

In this subsection we are going to investigate the general case of Definition 52.

Remark. Since WCP depends on the internal state of the object, the principle of *information hiding* can only be supported to a certain degree if we want to augment the methods of objects with worst-case timing information.

For example, if the WCP estimate of some operations of a tree package is $O(\log N)$, where N denotes the number of nodes in the tree, it is clear, that a balanced tree structure is used.

In a similar way we define for the stack space behavior of objects.

Definition 53. A *Worst-Case Space (WCS)* estimate for a certain method $\mathcal{M}_i$ of object $\mathcal{O}$ is a function from the set of tuples of parameters of the method ($\mathcal{F}_i$), the current internal state of $\mathcal{O}$ and the implementation details of $\mathcal{M}_i$ to a certain number of bits or bytes

$$\mathsf{WCS} : \mathcal{F}_i \times \mathcal{S}(\mathcal{O}) \times \mathcal{I}(\mathcal{M}_i) \rightarrow \mathbf{N}.$$

In the following, we will discriminate between two important cases:

1. The internal state of object $\mathcal{O}$ can be described by *generic parameters* that are *constant* during the life-time of $\mathcal{O}$.

2. The internal state of object $\mathcal{O}$ is reflected by some parameters. The value of such a *state parameter* may change if the internal state of $\mathcal{O}$ changes because of a method call. We suppose that methods change the internal state of the object *atomically*, i.e., while the code of a certain method executes, it is not possible to retrieve the value of a state parameter.

In the first case we speak of *generic worst-case performance* estimates (gWCP), in the second one of *actual worst-case performance* (aWCP).

Remark. In this context compare [Meyer, 1988], where so-called *class invariants* are defined. Generic parameters can be viewed as *object invariants*, i.e., properties being set at object creation time, but being fixed during the life-time of the object.

Definition 54. Let $\mathcal{P}_i$ denote the set of all possible values of the ith state parameter of object $\mathcal{O}$, $1 \leq i \leq m$.

Definition 55. Assume that all $\mathcal{P}_i$ are ordered. Let the ordering relation be denoted by "$<$" and write "$\leq$" if the operands are equal or the first one is smaller than the second one. Let $\overline{\mathbf{p}} = (p_1, \ldots, p_m), p_1 \in \mathcal{P}_1, \ldots, p_m \in \mathcal{P}_m$ denote a vector of values of state parameters. Let $\overline{\mathbf{P}}$ denote the set of all such vectors.

We define $\overline{\mathbf{p}}_1 \leq \overline{\mathbf{p}}_2$ if for all $1 \leq i \leq m$ we have $p_{1,i} \leq p_{2,i}$ componentwise, and $\overline{\mathbf{p}}_1 < \overline{\mathbf{p}}_2$ if $\overline{\mathbf{p}}_1 \neq \overline{\mathbf{p}}_2$ and $\overline{\mathbf{p}}_1 \leq \overline{\mathbf{p}}_2$.

Remark. Mentioning Proposition 49, we can suspect that both generic and state parameters are natural numbers. Thus deriving the required ordering relations should be straight-forward.

Definition 56. Let $\overline{\mathbf{p}}_1, \overline{\mathbf{p}}_2 \in \overline{\mathbf{P}}$. If $\overline{\mathbf{p}}_1 \leq \overline{\mathbf{p}}_2$ implies that $\mathsf{aWCP}(\overline{\mathbf{p}}_1) \leq \mathsf{aWCP}(\overline{\mathbf{p}}_2)$ for all methods $\mathcal{M}_i$, $1 \leq i \leq e$, we call the underlying object $\mathcal{O}$ *time-monotonical*.

We obtain the following theorem.

Theorem 57. *Worst-case performance estimates of time-monotonical objects can be ordered according to the following hierarchy*

$$\mathsf{aWCP} \leq \mathsf{gWCP}.$$

Again, no proofs are given. They can be found in [Blieberger, 1995].

Remark. Theorem 57 shows that in general aWCP estimates are sharper than gWCP estimates. aWCP estimates, however, are more difficult to obtain than gWCP estimates.

The same definitions and results can be applied to WCS estimates and the same arguments apply to worst-case space estimates. Details are left to the reader.

4.4.2 Augmenting Programming Languages with WCP and WCS Estimates

In this subsection we investigate how WCP and WCS estimates can be added to (real-time) programming languages. As a vehicle to illustrate our ideas, we use Ada95 (see [Ada, 1995]) because this language already incorporates features we can exploit in augmenting it with WCP estimates. However it is important to note that each programming language can be augmented with our features.

Generic Worst-Case Performance Estimates

In order to augment Ada95 with gWCP estimates, we can exploit generic packages and generic subprograms already present in Ada95. This feature even provides us with a generic parameter mechanism.

Actual Worst-Case Performance Estimates

State parameters, however, have to be added to the language. Several possibilities come to mind:

1. State parameters are identified with *generic out parameters*. Although Ada95 supports generic *in* and *in out* parameters, it does not support generic *out* parameters.
   Note that a generic package can either be a type manager or an object manager. In case of a type manager, binding of the state parameters to the generic packages is a bad idea. For object managers it works well.
2. The programmer of a real-time object must be forced to implement a primitive operation which returns the state parameter of the object.
   There are two possibilities:
   (a) we have to fix a certain name for this operation or
   (b) the programmer has to extend a predefined abstract data type, say `real_time`, which provides this operation.
   Approach (2a) suffers from the fact that the actual (discrete) type of the state parameter is not known in advance.
   Approach (2b) has several disadvantages:
   - It artificially increases the number of tagged types.
   - The actual type of the state parameter is not known in advance. Thus the tagged type has to be enveloped into a generic package.
   - Late binding, which is implied by tagged types, leads to overestimating the performance because one has to choose the maximum of the WCP of all possible calls or additional knowledge (on the algorithms used and the processed data) has to be incorporated into the program.

   Nevertheless Approach (2b) works well if tagged types and polymorphism are no problem for the real-time application. It is described in Section 4.6 in more detail.
3. State parameters are identified with some sort of discriminants. This gives the advantage that discriminants are readable parts of an object even if the type of the object is (limited) private. The only "uncommon" feature of state parameters is that they are—in contrast to discriminants—not constant.

In addition, it has to be ensured that state parameters cannot be read while a method call is executing. This can be guaranteed if the internal representation of the state parameter is a protected type. For details refer to Section 4.5.

Real-Time Methods

Each method $\mathcal{M}_i$ of a real-time object has to be augmented with functions computing gWCP and aWCP estimates. For most cases, it is enough to have one of these functions.

In addition, for aWCP estimates information has to be provided, how the state parameters change when the method is executed. Usually, there will be several possible cases.

4.4.3 Example

As an example we study a balanced tree structure such as a 2-3-tree, an AVL-tree, or a BB[α]-tree (see [Mehlhorn, 1984b]). Especially we are interested in inserting data into the tree. Let the number of nodes present in the tree be denoted by R and let an upper bound of the number of nodes be denoted by N.

Thus N plays the role of a generic parameter and R plays the role of a state parameter.

Figure 9 shows a specification of a corresponding Ada95 package. For didac-

```
generic

   N: in natural;
      -- generic parameter
   type item_type is private;

package Balanced_Tree is

   type tree(R: out natural) is private;                    -- out discriminant

   item_already_present: exception;

   procedure Insert(
      item: item_type,
      into: tree)
         with duration ≤ log(N) and                                -- gWCP
         with duration ≤ log(R)                                    -- aWCP
         new R := R+1 | R (item_already_present);
            -- R+1 ... if item has been inserted
            -- R   ... if item is already present (exception)

   -- other methods suppressed

end Balanced_Tree;
```

Figure 9: Specification of Balanced Tree Insertion

tic purposes we have assumed rather simple gWCP and aWCP estimates, which for example are not correct in the case $R = 0$. The state parameter is incremented if the item has been correctly inserted into the tree. It is left unchanged if the item has already been present in the tree. In this case the exception `item_already_present` is raised.

Now assume that an application of an instance of this package inserts N items one after the other. In terms of gWCP estimates, this means an overall

estimate of

$$N \log N.$$

In terms of aWCP estimates, we obtain (see e.g. [Graham et al., 1989])

$$\sum_{R=1}^{N} \log R = N \log N - N + \frac{\log N}{2} + \frac{1}{2} \log 2\pi + O(1/n).$$

Obviously this is a sharper estimate and more valuable for real-time applications.

It is the task of the compiler to validate the gWCP and aWCP estimates if the corresponding bodies (implementation parts) are compiled. In our example, if a discrete loop (see Section 2 and [Blieberger, 1994]) is used for implementing procedure `insert`, this can easily be done at compile time. Of course the details will be more complicated because we have only used simplified WCP estimates.

4.4.4 Syntactical and Semantical Issues

Although the syntax of WCP estimates is certainly important, we consider the semantical issues more important. In order to be able to demonstrate the advantages of WCP estimates, however, we define an Ada-like syntax. But it is important to note that an appropriate syntax can be defined for other languages too.

The syntax of WCP estimates is given by a notation similar to that of the *Ada Language Reference Manual* ([Ada, 1995]).

```
subprogram_declaration ::= subprogram_specification;
subprogram_specification ::=
      procedure defining_program_unit_name
            parameter_profile worst_case_specification
    | function defining_designator
            parameter_and_result_profile worst_case_specification
worst_case_specification ::=
      wc_performance_specification wc_space_specification
      [state_parameter_changes]
wc_performance_specification ::=
      gWCP_specification
            [and aWCP_specification] | aWCP_specification
wc_space_specification ::=
      gWCS_specification
            [and aWCS_specification] | aWCS_specification
gWCP_specification ::= with duration <= expression
      [and >= expression]
aWCP_specification ::= with duration <= expression
      [and >= expression]
gWCS_specification ::= with space <= expression
      [and >= expression]
aWCS_specification ::= with space <= expression
      [and >= expression]
state_parameter_changes ::=
      state_parameter_change { ; state_parameter_change }
```

```
state_parameter_change ::=
      new state_parameter := state_parameter_new_values
state_parameter_new_values ::=
      state_parameter_new_value { | state_parameter_new_value }
state_parameter_new_value ::=
      expression [(exception_name{,exception_name})]
```

We call a subprogram augmented with time or space estimates a *time* or *space constrained subprogram*. In contrast to [Ada, 1995] we allow discriminants of mode *out* and generic *out* parameters. Both are considered to be state parameters. A detailed description of the semantics of out discriminants and generic out parameters can be found in Section 4.5. The expressions of gWCP and aWCP are of type **duration** and the expressions of gWCS and aWCS are of type **natural**.

The expressions of gWCP and gWCS are allowed to consist of generic parameters, of *in* and *in out* parameters of the constrained subprogram, and of constants. If an in out parameter is used, the estimate is calculated by using the value of the parameter that is passed to the procedure and not the value that is returned after the procedure has been executed or any intermediate value.

The expressions of aWCP and aWCS are allowed to consist of generic and state parameters, of *in* and *in out* parameters of the constrained subprogram, and of constants.

The task of the compiler to verify the specified WCP estimates can be facilitated by restricting the syntax of the expressions of the WCP estimates. For example the compiler may allow the usual operators such as `+,-,*,/`...and several mathematical functions such as `log, exp, sqrt,`.... In addition sums, and min and max operators can be permitted. Then one can specify the following expression

```
sum(i, 1, N, log(i) + max(3*i, 2*i+5) )
```

where `N` is a generic parameter.

The *state_parameter_changes* part can be omitted if the internal state reflected by the state parameters does not change when the method is called or if there are no state parameters defined. Nevertheless, in the first case it is checked that the state has not been changed during the execution of the method call.

Before the code of a method M is executed, the expressions of the corresponding state parameter changes parts are evaluated. This results in a list of possible successive values of the state parameter. After the code of method M has been executed, it is checked whether the current value of the state parameter is contained in the list of possible successive values and if the method has been terminated by an exception, whether this exception is associated with the current state parameter. If one of these checks fails, the exception **state_error** is raised.

Which compile time and runtime checks can be performed is described in Section 4.7.

The remaining semantics should be clear from the previous discussion and from the syntactical description. Nevertheless we point out some important aspects:

- A constrained subprogram need not be a primitive operation of some type; in fact any subprogram can be constrained. The only difference is that a

subprogram not being a primitive operation must not have a state parameter changes part.
The only exception are operations of an object manager being implemented by a generic package; these operations are no primitive operations but are allowed to have a state parameters changes part.

- A constrained subprogram cannot use an unconstrained subprogram to implement its body.
- Goto statements must not be used for implementing constrained subprograms.
- The control flow of a constrained subprogram must not depend on global variables. This has to be checked by the compiler by doing some sort of control and data flow analysis.
- Generic tagged parameters require additional syntactical and semantical considerations. Assume a tagged type `t` and a primitive operation `o` given in Fig. 10. Then one can define a generic package `g` shown in Fig. 11. In order to enable this syntax, we need two attributes. Assume the predefined types depicted in Fig. 12. For `o` being a primitive operation, we define two attributes `o'Duration` and `o'Space` which return `duration_bounds` and `space_bounds` respectively.
  If `o`'s WCP estimate does depend on its parameters, parameters can be used for specifying `g_o`'s WCP, i.e., for example one can write

  ```
  with duration <= ... o(v=>a)'Duration.upper ...
  ```

  where `a` is some actual parameter of type `t`.

type t **is** [**abstract**] **tagged** ...;

procedure o(v: t)
  with duration $\leq$ ...;

Figure 10: A Tagged Type and a Primitive Operation

4.5 Semantics of Out Discriminants and Generic Out Parameters

Two different ways of constructing object-oriented agents are possible:

- type manager and
- object manager.

For each of these we provide an appropriate solution. Type managers are built with help of out discriminants of the underlying type and object managers are built with help of generic out parameters of generic packages.

As already stated reading of a state parameter cannot result in race conditions, but two successive read operations may return different values if an intermediate call (of another task) has changed the state of the object such that

```
generic
   type e_t is new t with private;
package g is
   procedure g_o( e_v: e_t)
      with duration ≤ ( ... o'Duration.upper ...);
      -- expression involving o's upper bound of duration

   ...

private
   ...
end g;
```

Figure 11: A Generic Package with a Generic Tagged Parameter

```
type duration_bounds is
   record
      lower, upper: duration;
   end record;

type space_bounds is
   record
      lower, upper: natural;
   end record;
```

Figure 12: Some Predefined Types

the out discriminant is affected too. Thus the two expressions (involving the out discriminant `N`) `H := N+N` and `H := 2*N` may deliver different results. The programmer is supposed to be cognizant of this problem.

Both, out discriminants and generic out parameters, must not be supplied with initial values.

4.5.1 Out Discriminants

Out discriminants can be mapped to ordinary record components which are enveloped in a protected type. In addition, a primitive operation returning the enveloped record component can be added to the set of primitive operations. Thus reading the out discriminant actually is a function call to this primitive operation which internally is forwarded to the protected object. An example how such a protected type looks like is given in the following.

Package `Protected_Types` is supposed to contain the generic specification and implementation of a protected type for discrete types (see Figures 13 and 14). The given operations are supposed to be used for setting and retrieving the value of the underlying out discriminant. Function `Read` is supposed for retrieving the value of the protected type, procedure `Set` is used for updating the value.

The reasons why state parameters can be supposed to be of discrete type or even of type natural, are given in Section 4.3.

```
generic
   type disc is (<>);
package Protected_Types is

   protected type protected_discrete is
      function Read return disc;
      procedure Set(to: in disc);
   private
      val: disc;
   end protected_discrete;

end Protected_Types;
```

Figure 13: Specification of Package `Protected_Types`

```
package body Protected_Types is

   protected body protected_discrete is

      procedure Set(to: in disc) is
      begin
         val := to;
      end set;

      function Read return disc is
      begin
         return val;
      end read;

   end protected_discrete;

end Protected_Types;
```

Figure 14: Implementation of Package `Protected_Types`

4.5.2 Generic Out Parameters

The semantics of generic out parameters differ from that of generic in out parameters significantly. The major differences are:

- Generic formal out parameters must not be supplied with generic actual parameters during instantiation.
- Generic out parameters cannot be written from outside, e.g. from the instantiating package.
- Inside the generic package generic out parameters can be read and written.
- As mentioned above, race conditions cannot occur when reading a generic out parameter.

Generic out parameters can be enveloped in protected objects just as it can be done for out discriminants. The generic out parameter itself can be mapped to

a function within the generic package. Again, reading the generic out parameter actually is a function call which internally is forwarded to the protected object.

4.6 An Abstract Type for Real-Time Objects

In this subsection we describe how object-oriented features can be used for providing objects with WCP estimates. The techniques presented in Section 4.5 can be used without tagged types, they rely only on generic parameters and on some sort of discriminants.

Note that the methods presented in this subsection can also be used for object-oriented programming languages which do not support generics and discriminants adequately.

Since the exact type of the state parameters is not known in advance, a generic package must be used for the declaration of the appropriate types `real_time_type` and `limited_real_time_type` (see Fig. 15). An instantiation of this

```
generic

   type state_param_type is private;

package Real_Time is

   type real_time_type is abstract tagged private;
   function State(obj: in real_time_type)
      return state_param_type is abstract;

   type limited_real_time_type is abstract tagged limited private;
   function State(obj: in limited_real_time_type)
      return state_param_type is abstract;

private
   type real_time_type is abstract tagged null record;
   type limited_real_time_type is abstract tagged limited null record;
end Real_Time;
```

Figure 15: Abstract Real-Time Types

package is shown in Fig. 16. Notice, however, that state parameters might be

```
with Real_Time;
package My_Real_Time is new Real_Time(state_param_type => natural);
```

Figure 16: An Instantiation

enumerations, natural numbers, and suitable combinations of these discrete types such as array or record structures.

From these abstract real-time types a programmer can derive his own (non-abstract) real-time types by extending it with whatever is needed and by implementing an appropriate `State` function (see Fig. 17). Note that this is very similar to the handling of controlled types.

Remark. Note that the programmer has to implement the body of function `State` in a way such that race conditions are impossible, e.g. by a protected object. This means that the whole responsibility on this issue is on the programmer while with the approaches of Section 4.5 absence of race conditions can be guaranteed by a (pre-)compiler.

`State` can be renamed and this name or `State` must be used for the state parameter changes part (see Section 4.4.4).

Note, however, that declaring a controlled real-time type poses some problems:

- A technique described in [Rat, 1995] has to be applied to exploit multiple inheritance.
- This technique uses access discriminants, which are only allowed for limited types.

Thus we can only use *limited* controlled real-time types, which means that we cannot use assignment for these types (see Fig. 18).

```
with My_Real_Time;
package My_Real_Time_Defs is
  type my_real_time_type is
    new My_Real_Time.limited_real_time_type with private;
  function State(obj: my_real_time_type) return natural;
private
  type my_real_time_type is
    new My_Real_Time.limited_real_time_type with
  record
    n: natural;
    -- other record components suppressed
  end record;
end My_Real_Time_Defs;
```

Figure 17: Some Useful Definitions

The remark and the other disadvantages mentioned above give reasons why the techniques described in Section 4.5 are useful, and they justify the language extensions proposed in Section 4.5.

4.7 Runtime and Compile Time Checks

In this subsection we discuss which checks can be performed at compile time and at runtime with help of our WCP estimates.

```
with My_Real_Time_Defs, Ada.Finalization;
package My_Real_Time_Controlled is
  type my_real_time_controlled_mixin(
       rt: access My_Real_Time_Defs.my_real_time_type'CLASS) is
    new Ada.Finalization.limited_controlled with private;
  type my_real_time_controlled_type is
       new My_Real_Time_Defs.my_real_time_type with
    record
      contr: my_real_time_controlled_mixin(
        my_real_time_controlled_type'ACCESS);
    end record;
private
  type my_real_time_controlled_mixin(
       rt: access My_Real_Time_Defs.my_real_time_type'CLASS) is
    new Ada.Finalization.limited_controlled with null record;
end My_Real_Time_Controlled;
```

Figure 18: The Limited Controlled Real-Time Type

4.7.1 Runtime Checks

The WCP estimate added to the specification of a subprogram can be used at runtime to ensure that the specified duration is not exceeded. Indeed there are several possible directions to proceed:

EST1 *The specified duration is compared to an estimate of the execution time provided by the compiler.*
The compiler calculates estimates for the execution time of all basic blocks of the machine code and "inserts" this estimate at the beginning of the basic block into the machine code. Compare [Harmon et al., 1994] for how such estimates can be derived in a retargetable way.
At runtime the following algorithm is performed:
- At the beginning of the execution of the constrained subprogram the specified duration is copied into a local variable `remaining_time`.
- At the beginning of each basic block the inserted estimate of the execution time is subtracted from `remaining_time`.
- If the value is still positive, the execution of the basic block continues; otherwise the exception **timing_error** is raised.

EST2 *The specified duration is compared to an estimate of the execution time provided by the compiler.*
In fact the same algorithm is performed as above, the only difference is that instead of estimates of the execution time of the basic blocks, the actual execution time of the basic block is used. This can be found by executing the basic block and measuring its execution time. This measured time is then inserted at the beginning of the basic block. Special care has to be taken that the execution time does not depend on data values occurring in the basic block. For example consider machine statements which consume a different number of machine cycles if different data is processed.
This approach can take into account pipelining of the target processor without problems. In contrast **EST1** has to cope with pipelining separately.

ACT *The specified duration is compared to the actual execution time of the subprogram.*
This means that the execution time of a constrained subprogram is measured and compared to the specified estimate after the subprogram has been executed. Note that this can also be done at the level of basic blocks, which results in earlier detection of **timing_error**.

COMB *A combination of **EST1** (**EST2**) and **ACT**.*
Before a basic block is executed, `remaining_time` is checked against an estimate which is calculated by approach **EST1** or **EST2**, but after the basic block has been executed its actual execution time is subtracted from `remaining_time`.

Note that **EST1** and **EST2** can even be performed if the target processor is simulated and the simulated processor is slower than the original target processor.

Remark. We have assumed that the scheduler does not preempt a basic block. Dispatching points are only at the beginning of a basic block.

4.7.2 Compile Time Checks

Clearly if the compiler is able to prove that the specified WCP estimate is correct, no runtime checks are needed. This proof comprises several tasks:

1. Show that the number of loop iterations are in accordance with the specified WCP estimate (see Section 2 and [Blieberger, 1994]).
2. Investigate recursive subprograms (if present) to ensure that they meet certain requirements (see Section 3 and [Blieberger and Lieger, 1994]).
3. Perform control and data flow analysis in order to show that the specified WCP estimate is correct.

Note that the compiler does not need to do all of the above proofs. In fact, the more the compiler can prove, the less runtime checks are necessary.

4.8 Concluding Remarks on Specifying Worst-Case Performance Estimates on the Level of Classes

Object-oriented features can be used for real-time applications. In fact, introducing object-oriented concepts and object-oriented programming into the design and implementation phases of real-time projects, provides real-time applications with considerable advantages in terms of reusability and maintainability.

If a real-time language is augmented with the features we have presented in this section, additional benefits can be derived. Providing WCP estimates at an object's interface enhances modularization and testing of the real-time system.

Most checks concerning the real-time property of the system can be done by a suitable compiler.

5 Conclusion

In this paper we have shown that programming language support for estimating the worst-case performance of real-time programs is possible and results in great improvements for real-time applications.

While almost all recent approaches have been "negative" by forbidding language features, our approach is "positive" in that we are introducing *new* language constructs for facilitating WCP estimates.

- We have presented a new kind of loops (discrete loops), which are especially useful for real-time applications since they can be used for applications which otherwise would have required general loop statements. The number of iterations of discrete loops, however, can be determined at compile time.
- We have shown that recursive procedures and functions can be used for real-time applications if a few simple conditions are met.
- Finally, we have augmented object-oriented languages such that estimating the worst-case performance of programs is facilitated in a significant manner.

We are currently implementing a precompiler which augments Ada95 with all these language features (see [Blieberger et al., 1996]). This work is part of Project WOOP (Worst-Case Performance of Object-Oriented Real-Time Programs) which is carried out at the *Department of Automation* at the *Technical University of Vienna*.

Acknowledgments

The author would like to thank the WOOP team for many fruitful discussions. Special thanks go to Roland Lieger, Bernd Burgstaller, Bernhard Scholz, and Mario Weilguni.

References

[Ada, 1995] Ada (1995). *Ada Reference manual*. ISO/IEC 8652.

[Aho et al., 1974] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

[Aho et al., 1983] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1983). *Data Structures and Algorithms*. Addison-Wesley, Reading, MA.

[Blieberger, 1994] Blieberger, J. (1994). Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212.

[Blieberger, 1995] Blieberger, J. (1995). Timing analysis of object-oriented real-time programs. (submitted).

[Blieberger and Lieger, 1994] Blieberger, J. and Lieger, R. (1994). Worst-case space and time complexity of recursive procedures. *Real-Time Systems*. (to appear).

[Blieberger et al., 1996] Blieberger, J., Lieger, R., and Burgstaller, B. (1996). Augmenting Ada95 with additional real-time features. In *Proc. of Ada-Europe'96*, Montreux. LNCS. (to appear).

[Blum and Mehlhorn, 1980] Blum, N. and Mehlhorn, K. (1980). On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320.

[Booch, 1991] Booch, G. (1991). *Object-oriented design with applications*. Benjamin/Cummings, Redwood City, CA.

[Businger, 1985] Businger, A. (1985). *PORTAL Language Description*, volume 198 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin.

[DIN, 1982] DIN (1982). *Programmiersprache PEARL, Full PEARL*. DIN 66 253, Teil 2, Beuth Verlag, Berlin.

[Ellis and Stroustrup, 1990] Ellis, M. A. and Stroustrup, B. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA.

[Forsyth, 1993] Forsyth, C. (May 1993). Using the worst-case execution analyser. Technical report, York Software Engineering Ltd., University of York: Task 8, Volume D Deliverable on ESTEC contract 9198/90/NL/SF.

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

[Graham et al., 1989] Graham, R. L., Knuth, D. E., and Patashnik, O. (1989). *Concrete Mathematics*. Addison-Wesley, Reading, MA.

[Gwinn, 1992] Gwinn, J. (1992). Object-oriented programs in realtime. *ACM SIGPLAN Notices*, 27(2):47–56.

[Halang and Stoyenko, 1991] Halang, W. A. and Stoyenko, A. D. (1991). *Constructing predictable real time systems*. Kluwer Academic Publishers, Boston.

[Harmon et al., 1994] Harmon, M. G., Baker, T. P., and Whalley, D. B. (1994). A retargetable technique for predicting execution time of code segments. *Real-Time Systems*, 7:159–182.

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

[Hofstadter, 1979] Hofstadter, D. R. (1979). *Gödel, Escher, Bach - an Eternal Golden Braid*. Basic Books, New York.

[Ishikawa et al., 1990] Ishikawa, Y., Tokuda, H., and Mercer, C. W. (1990). Object-oriented real-time language design: Constructs for timing constraints. In *ECOOP/OOPSLA '90 Proceedings*, pages 289–298.

[Kligerman and Stoyenko, 1986] Kligerman, E. and Stoyenko, A. D. (1986). Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949.

[Knuth, 1973a] Knuth, D. E. (1973a). *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., second edition.

[Knuth, 1973b] Knuth, D. E. (1973b). *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass.

[Knuth, 1981] Knuth, D. E. (1981). *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., second edition.

[Kopetz et al., 1989] Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., and Zainlinger, R. (1989). Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40.

[Li and Malik, 1995] Li, Y.-T. S. and Malik, S. (1995). Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, volume 30, La Jolla, California. ACM SIGPLAN Notices.

[Lieger and Blieberger, 1994] Lieger, R. and Blieberger, J. (1994). The Ackermann-function effort in space and time. Technical Report 183/1-48, Department of Automation, Technical University Vienna.

[Liu and Layland, 1973] Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61.

[Maclean and Smith, 1995] Maclean, S. and Smith, S. (1995). Object-oriented programming for embedded systems. *ACM SIGPLAN Notices*, 30(9):3–8.

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

[Mehlhorn, 1984b] Mehlhorn, K. (1984b). *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, Berlin.

[Meyer, 1988] Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice Hall, New York, NY.

[Mok, 1984] Mok, A. K. (1984). The design of real-time programming systems based on process models. In *Proceedings of the IEEE Real Time Systems Symposium*, pages 5–16, Austin, Texas. IEEE Press.

[Mok et al., 1989] Mok, A. K., Amerasinghe, P., Chen, M., and Tantisirivat, K. (1989). Evaluating tight execution time bounds of programs by annotations. In *Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80.

[Nievergelt and Reingold, 1973] Nievergelt, I. and Reingold, E. (1973). Binary search trees of bounded balance. *SIAM Journal of Computing*, 2(1):33–43.

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

[Park, 1993] Park, C. Y. (1993). Predicting program execution times by analyzing static and dynamic program paths. *The Journal of Real-Time Systems*, 5:31–62.

[Puschner and Koza, 1989] Puschner, P. and Koza, C. (1989). Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1:159–176.

[Rat, 1995] Rat (1995). *Ada95 Rationale*. Intermetrics, Inc.

[Schaffer and Sedgewick, 1993] Schaffer, R. and Sedgewick, R. (1993). The analysis of heapsort. *Journal of Algorithms*, 15:76–100.

[Schedl, 1993] Schedl, A. V. (1993). *Zeitanalyse von Echtzeitprogrammen mittels linearer Optimierung*. Diplomarbeit, Inst. f. Technische Informatik, TU Vienna, Austria.

[Sedgewick, 1988] Sedgewick, R. (1988). *Algorithms*. Addison-Wesley, Reading, MA, second edition.

[Shaw, 1989] Shaw, A. C. (1989). Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889.

[Stoyenko and Marlowe, 1992] Stoyenko, A. D. and Marlowe, T. J. (1992). Polynomial-time transformations and schedulability analysis of parallel real-time programs with restricted resource contention. *The Journal of Real-Time Systems*, 4:307–329.

[Vitter and Flajolet, 1990] Vitter, J. S. and Flajolet, P. (1990). Average-case analysis of algorithms and data structures. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 431–524. North-Holland.

[Vrchoticky, 1993] Vrchoticky, A. (May 1993). Modula/R language definition. Technical report, Inst. für Technische Informatik, TU Vienna, Austria.

[Xu and Parnas, 1993] Xu, J. and Parnas, D. L. (1993). On satisfying timing constraints in hard real-time systems. *IEEE Trans. Software Engineering*, 19(1):70–84.