Loops for Safety Critical Applications*

Johann Blieberger
Department of Automation, Technical University Vienna
Vienna, Austria

Abstract

In this paper so-called *discrete loops* are described which narrow the gap between general loops (e.g. while- or repeat-loops) and for-loops. Although discrete loops can be used for applications that would otherwise require general loops, discrete loops are known to complete in any case. Furthermore it is possible to determine the number of iterations of a discrete loop, while this is trivial to do for for-loops and extremely difficult for general loops. Thus discrete loops form an ideal frame-work for determining the worst case timing behavior of a program and they are especially useful in implementing real-time and safety related systems and proving such systems correct.

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 considered for implementing those loops that can not be handled by for-loops. These loops include while-loops, repeat-loops, and loops with exit-statements (cf. e.g. [Ada95]).

If a general loop does not complete, the corresponding program usually is incorrect. It is possible to use some logical devices, such as *Hoare logic* (cf. [Hoa69]), to prove that a certain loop (and thus the corresponding program) completes. This however implies that the programmer has to be an expert in formal logics. In most cases, however, programmers convince themselves by testing that their loops complete.

Concerning real-time systems the program behavior must not only be correct but the result of a computation must be available within a predefined deadline. It has turned out that a major progress in order to guarantee the timeliness of real-time systems can only be achieved if the *scheduling problem* is solved accordingly. Most scheduling algorithms assume that the runtime of a task is known a priori (cf. e.g. [LL73, HS91, Mok84]). Thus the *worst case performance* of a task plays a crucial role.

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

The most difficult task in estimating the timing behavior of a program is to determine the number of iterations of a certain loop.

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.

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. [HS91, PK89, NP93, Par93]. Most researchers, however, 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 the loop has to complete (cf. e.g. [ITM90]). In any case the user, i.e., the programmer, has to react to such exceptional cases.

In this paper we will narrow the gap between general loops and for-loops by defining *discrete loops.* These loops are known to complete and are easy to analyze (especially their number of iterations) and capture a large part of applications which otherwise would have been implemented by use of general loops.

Clearly, discrete loops form an ideal frame-work for determining the worst case timing behavior of a program and they are especially useful in implementing real-time and safety related systems and proving such systems correct.

2 Discrete Loops

In this section we give an informal introduction to discrete loops, before we perform a theoretical treatment, i.e., an exact definition and some mathematical results. Further results can be found in [Bli94].

2.1 Introduction to Discrete Loops

In contrast to for-loops, discrete loops allow for 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.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**

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

Figure 1: A simple example of a discrete loop

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.

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 := 1 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 is definitely 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.

By the way, a loop like that in Figure 2 occurs in a not-recursive implementation of *Heapsort* (cf. [Knu73] or [SS93] for a more readable form in a high-order programming language).

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.

2.2 Theoretical Treatment

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

Definition 2.2. 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 2.1. Note that $k_\nu \in \boldsymbol{N}$ for all $\nu \in \boldsymbol{N}$.

Definition 2.3. 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 $\omega \in \boldsymbol{N}$. 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 2.4. 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 $s \in [1, N]$.

3 Monotonical Discrete Loops

Definition 3.1. A sequence (k_ν) is called *strictly monotonically increasing* if $k_{\nu+1} > k_\nu$ for all $\nu \geq 1$. It is called *strictly monotonically decreasing* if $k_{\nu+1} < k_\nu$ for all $\nu \geq 1$.

Definition 3.2. 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 3.1. *A monotonical discrete loop is completing.*

Proof. If all (k_ν) are strictly monotonically increasing, there certainly must exist some $\omega \geq 1$ such that $k_\omega \leq N < k_{\omega+1}$. Thus the loop completes.

On the other hand, if all (k_ν) are strictly monotonically decreasing, there certainly must exist some $\omega \geq 1$ such that $k_\omega \geq 1 > k_{\omega+1}$. Thus the loop completes in this case too.

Lemma 3.2. *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 $x \in [1, N]$.

Proof. If there would exist some f_d such that $f_d(x) \leq x$, there would exist an iteration sequence (k_ν) such that $k_{\nu+1} = f_d(k_\nu) \leq k_\nu$ which contradicts Definition 3.2.

Lemma 3.3. *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 $x \in [1, N]$.

Proof. If there would exist some f_j such that $f_j(x) \geq x$, there would exist an iteration sequence (k_ν) such that $k_{\nu+1} = f_j(k_\nu) \geq k_\nu$ which contradicts Definition 3.2.

3.1 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 [Ada95].

```
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., it can be used on both sides of an assignment statement for example.

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 3.1. 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 fail, i.e., one of the exceptions **monotonic_error** or **successor_error** is raised.

Remark 3.2. 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.

4 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.

4.1 Lower and Upper Bounds

Definition 4.1. 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

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

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.

In order to calculate U and L we can use algorithms given in [Meh84a]. The following Theorem 4.1, however, will show that under certain conditions U and L can be determined much easier. Before that we need one further definition.

Definition 4.2. 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 4.1. *If for all $1 \leq i \leq e$ $f_i(1) > 1$ and $f_i(x+1) - f_i(x) \geq 1$ for all $x \in \boldsymbol{N}$, then*

1. *the corresponding discrete loop completes,*
2. *the length of $(k_\nu^{(\max)})$ is equal to* L, *and*
3. *the length of $(k_\nu^{(\min)})$ is equal to* U.

A proof of Theorem 4.1 can be found in [Bli94].

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 4.1 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.

4.2 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 $N \in \boldsymbol{N}$ 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 4.2. *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*

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

Proof. We clearly have

$$f(x) = \lceil \alpha x + \beta \rceil \geq \alpha x + \beta.$$

Thus

$$k_\nu \geq \alpha^{\nu-1} + \frac{\alpha^{\nu-1}-1}{\alpha-1}\beta = \alpha^{\nu-1}\left(\frac{\alpha+\beta-1}{\alpha-1}\right) - \frac{\beta}{\alpha-1}.$$

To estimate len k_ν we must have

$$\alpha^{\nu-1}\left(\frac{\alpha+\beta-1}{\alpha-1}\right) - \frac{\beta}{\alpha-1} > N$$

which is equivalent to

$$\alpha^{\nu-1} > \frac{N(\alpha-1)}{\alpha+\beta-1} + \frac{\beta}{\alpha+\beta-1}.$$

Taking logarithms we have proved the theorem.

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

Integrating the results of Theorem 4.2 and similar theorems into a compiler, the number of iterations of discrete loops can often be estimated at compile time.

5 Discrete Loops with a Remainder Function

Definition 5.1. 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 5.1. Definition 5.1 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. For example in *binary search* the number of the remaining items is equal to the length of the remaining interval.

Definition 5.2. A loop sequence of remaining items is called *monotonical* if $r_{\nu+1} < r_\nu$.

Definition 5.3. A discrete loop with a remainder function is called *monotonical* if all its loop sequences $(r_\nu) \in \mathcal{R}$ are monotonical.

Lemma 5.1. *A monotonical discrete loop with a remainder function is completing.*

Proof. Since a monotonically decreasing discrete function will become smaller than 1 in finitely many steps, the corresponding loop will complete.

5.1 Syntactical and Semantical Issues of Discrete Loops with Remainder Functions

The syntax of a discrete loop with a remainder function is again given by a notation similar to that in [Ada95]. In fact we add to the syntax definition of Section 3.1.

```
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 ::=
    monotonical_discrete_loop_parameter_specification |
    discrete_loop_with_remainder_function_parameter_specification

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

discrete_loop_with_remainder_function_parameter_specification ::=
    [identifier := initial_value
        new identifier := list_of_iteration_functions]
        with rem_identifier := initial_value new remainder_function

list_of_iteration_functions ::=
    iteration_function { | iteration_function }

iteration_function ::= expression

remainder_function ::=
    rem_identifier =   expression |
    rem_identifier <= expression [ and rem_identifier >= expression ]
```

For a discrete loop with a remainder function, the corresponding loop parameter specification is the optional declaration of the *loop variable* with the given identifier. The loop variable is an object whose type is the base type of result type of the iteration functions, which must be the same for all iteration functions. The initial value of the loop variable is given by initial_value. Within the sequence of statements the loop variable behaves like any other variable, i.e., it can be used on both sides of an assignment statement for example.

After the keyword **with** the *remainder loop variable* is declared by the given identifier (*rem*_identifier). Its type must be a subtype of **natural** in the cases (1) and (2) below or an interval between two **natural** numbers in the case (3). Its initial value is given by initial_value. The remainder function itself may have three different forms:

1. If the remainder function can be determined exactly, it is given by an equation.

2. If only an upper bound of the remainder function is available, it is given by an inequality (<=).

3. If in addition to (2) a lower bound of the remainder function is known, it can be given by an optional inequality (>=). The second inequality must be separated from the first one by the keyword **and**.

The base type of the expressions defining the remainder function or its bounds must be **natural**.

In case (1) the remainder loop variable behaves like a constant within the sequence of statements. In cases (2) and (3) the remainder loop variable behaves like any other variable within the sequence of statements. If the value of the remainder loop variable is changed during the execution of the statements, we call the original value *previous value* and the new value *current value*.

Before the sequence of statements is executed, the list of iteration functions is evaluated if a loop variable is given. This results in a list of *possible successive values*.

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.

After the sequence of statements has been executed, the remainder function or its bounds (depending on which are given by the programmer) are evaluated. In case (1) the new value of the remainder loop variable is set to the value calculated by the remainder function if it is smaller than the previous value, otherwise the exception **monotonic_error** is raised.

In case (2) the new value of the remainder loop variable is set to the value calculated by the remainder function if the previous value of the remainder loop variable is equal to its current value and if the calculated value is smaller than the current value, otherwise the exception **monotonic_error** is raised. If the previous and the current value differ, the remainder loop variable is set to the current value if it is smaller than or equal to the calculated value, which in turn must be smaller than the previous value. If this is not true, the exception **monotonic_error** is raised.

In case (3), at the beginning both the lower and upper bound of the remainder loop variable are set to the initial value provided by the programmer. After the loop body has been executed the new upper and lower bounds of the remainder loop variable are set to the values calculated by the appropriate remainder functions if the current value of the remainder variable is equal to the previous value and if the calculated upper bound is smaller than the current value of the upper bound and if the calculated lower bound is smaller or equal to the current value of the lower bound. If the current value and the previous value differ, both the upper and lower bound are set to the current value if the current value is smaller than the calculated upper bound, which in turn must be smaller than the previous upper bound, and if the current value is greater than the calculated lower bound, which in turn must be smaller or equal than the previous lower bound. Otherwise the exception **monotonic_error** is raised. This exception is raised too if the interval does not contain at least one element.

If in cases (1) and (2) the value of the remainder loop variable is zero or if in case (3) the upper bound is zero, the exception **loop_error** is raised, otherwise the loop is continued.

The regular way to complete a discrete loop with a remainder function is to use an *exit* statement, before the remainder loop variable is equal to zero.

Remark 5.2. The semantics of discrete loops with remainder functions ensure that such a loop will always complete, either if the loop is terminated by an *exit* statement or because one of the above check fails, i.e., one of the exceptions **monotonic_error**, **successor_error**, or **loop_error** is raised.

Remark 5.3. A corresponding compiler is free to perform as many checks as it likes in order to inhibit one of the runtime exceptions **monotonic_error**, **successor_error**, and **loop_error**. This can be done by ensuring that the remainder function or its bounds are monotonical, by performing data-flow analysis to make sure that **successor_error** will never be raised, or by ensuring that the loop will complete before the remainder loop variable is equal to zero. Thus a lot of runtime checks can be avoided.

Moreover the compiler might even detect bounds of the number of iterations of the loop, which is a valuable result for real-time applications.

5.2 Some Examples of Monotonical 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 3. In this figure `root` denotes a pointer to the root of the tree, `height` 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.

```
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 if node_pointer = null.

end loop;
```

Figure 3: Template for Traversing Binary Trees

Weight-Balanced Trees

So-called *weight-balanced trees* have been introduced in [NR73] and are treated in detail in [Meh84b].

Definition 5.4. 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 [Meh84b]).

BB[α]-trees are binary trees with bounded height. In fact it is proved in [Meh84b] 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 4, where **floor(x)** is supposed to implement $\lfloor x \rfloor$. The remainder function of Figure 4 has the advantage that it does not need logarithms since it works with the number of leaves instead of the height of the tree.

```
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 4: Another Template for Operations on BB[α]-trees

5.3 The Number of Iterations of a Monotonical Discrete Loop with a Remainder Function

Theorem 5.1. *If a loop sequence of remaining items fulfills*

$$r_1 = N,$$

$$r_{\nu+1} = \lfloor r_\nu / \mu \rfloor,$$

where $\mu > 1$, then len r_ν is bounded above by

$$\lfloor \log_\mu N + 2 \rfloor.$$

Proof. We clearly have

$$\lfloor r_\nu / \mu \rfloor \leq r_\nu / \mu.$$

Thus

$$r_\nu \leq \frac{N}{\mu^{\nu-1}}$$

and to estimate the length of (r_ν) we must have

$$N < \mu^{\nu-1}.$$

Taking logarithms the theorem is proved.

6 Discrete Loops and Safety

There are several reasons why safety related systems can profit from discrete loops:

- The syntax and semantics of discrete loops are easy enough to permit validation or even verification of a suitable compiler.

  This is especially true if only runtime checks are to be performed, i.e., no compile time checks such as solving recurrences or data-flow analysis.

- Since discrete loops are known to complete in any case, *no* endless loop can occur.

  Thus during verification or validation no effort has to be spent in order to prove that the application will complete. (We assume that tasks can be scheduled periodically.) Note that this can be done without having to rely on formal logical devices such as *Hoare Logic* (cf. [Hoa69, LS87]).

- Since the number of iterations of a discrete loop is bounded from below and from above, it is easy to derive lower and upper bounds for the timing behavior of an application. Even if no automated tool can be used for that purpose, information on the timing behavior can be derived by hand.

  Thus the validation process can provide exact bounds for the timing behavior of the application, again without use of formal logics such as in [Sch92]. This is a valuable basis to start schedulability analysis.

- Since all important steps can be done by automated tools, validating or even verifying these tools can save validation effort for applications. Such tools include compilers and schedulability analyzers (cf. e.g. [HS91]).

Of course in the discussion above we have assumed that no while loops are present in the application to be validated or verified.

7 Conclusion

In this paper we have described discrete loops which narrow the gap between general loops and for-loops. Since they 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 and safety related applications.

Thus we conclude that only for-loops and discrete loops should be allowed for implementing safety related and real-time systems.

Development of a precompiler implementing discrete loops is part of Project WOOP which is carried out at the *Department of Automation* at the *Technical University of Vienna.*

References

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

[Bli94] J. Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12:576–580, 1969.

[HS91] W. A. Halang and A. D. Stoyenko. *Constructing predictable real time systems*. Kluwer Academic Publishers, Boston, 1991.

[ITM90] Y. Ishikawa, H. Tokuda, and C. W. Mercer. Object-oriented real-time language design: Constructs for timing constraints. In *ECOOP/OOPSLA '90 Proceedings*, pages 289–298, October 1990.

[Knu73] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., 1973.

[LL73] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LS87] J. Loeckx and K. Sieber. *The foundations of program verification*. Wiley-Teubner Series in Computer Science. John Wiley & Sons and B.G. Teubner, New York and Stuttgart, second edition, 1987.

[Meh84a] K. Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms*. Springer-Verlag, Berlin, 1984.

[Meh84b] K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, Berlin, 1984.

[Mok84] A. K. Mok. 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, 1984. IEEE Press.

[NP93] V. Nirkhe and W. Pugh. A partial evaluator for the Maruti hard real-time system. *The Journal of Real-Time Systems*, 5:13–30, 1993.

[NR73] I. Nievergelt and E. Reingold. Binary search trees of bounded balance. *SIAM Journal of Computing*, 2(1):33–43, 1973.

[Par93] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *The Journal of Real-Time Systems*, 5:31–62, 1993.

[PK89] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.

[Sch92] D. J. Scholefield. *A Refinement Calculus for Real-Time Systems*. PhD thesis, University of York, 1992.

[SS93] R. Schaffer and R. Sedgewick. The analysis of heapsort. *Journal of Algorithms*, 15:76–100, 1993.