Augmenting Ada95 with Additional Real-Time Features*

Johann Blieberger, Roland Lieger, and Bernd Burgstaller

Department of Automation (183/1),
Technical University of Vienna,
Treitlstr. 1/4, A-1040 Vienna
email: {blieb,rlieger,bburg}@auto.tuwien.ac.at

Abstract. The purpose of this paper is to present several additions to Ada95 which improve real-time properties of the language. 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.

1 Introduction

The most significant difference between real-time systems and other computer systems is that the system behavior must not only be correct but 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. Thus the *worst-case performance* of a task plays a crucial role.

The most difficult tasks in estimating the timing behavior of a program are to determine the number of iterations of a certain loop and to handle problems originating from the use of recursion. This paper presents our solutions to these problems by augmenting Ada95 with appropriate language features.

Currently we are building a precompiler for Ada95 which implements these language features. Its output is again Ada95 and the precompiler is actually built by modifying GNAT, the Gnu Ada Translator (cf. [SB94]).

2 Discrete Loops

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.

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

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]).

Determining the number of iterations of a for-loop is trivial. 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, Par93, PK89].

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 the loop has to complete. In any case the user, i.e., the programmer, has to react to such exceptional cases.

Our approach is different in that we define a new kind of loops, so-called discrete loops, which are described in detail in [Bli94]. In contrast to for-loops, discrete loops allow for a more complex dependency between two successive values of the loop-variable.

Like for-loops discrete loops have a loop-variable and an integer range associated with them. The major difference to for-loops is that the loop-variable need not be assigned each value 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.

An example for a discrete loop is depicted in Fig. 1. 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 $1, 3, 6, 13, 26, 52, 105, \ldots$.

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

Fig. 1. An example of a discrete loop

Another form of discrete loops are *discrete loops with a remainder function*. These loops are especially well-suited for algorithms designed to traverse binary trees and for *divide & conquer algorithms*. A template showing such applications is given in Fig. 2. 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
end loop;
```

Fig. 2. Template for Traversing Binary Trees

In any case the number of iterations of discrete loops can be determined at compile time if the iteration functions are monotonic (cf. [Bli94]).

3 Real-Time Recursions

In view of the problems that arise when recursions are to be used in real-time applications, most designers of real-time programming languages decide to forbid them in their languages, e.g. RT-Euclid, PEARL, Real-Time Concurrent C, and the MARS-approach.

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.

Our approach is different in that we do not forbid recursion, but instead constrain recursive procedures 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.

Definition 1. 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.

Definition 2. 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.

Definition 3. We denote by $f_1 \prec f_2$ a binary relation, which implies that the recursion depth of the underlying recursive procedure applied to f_1 is smaller than if applied to f_2. If the recursion depth of f_1 is smaller than or equal to that of f_2, we write $f_1 \preceq f_2$.

The set $\mathcal{F}_i$ contains all $f \in \mathcal{F}$ with recursion depth i.

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

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

Definition 4. 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 $\tau(f_1) \leq \tau(f_2)$.

Definition 5. 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 $m_1 \leq m_2$ and $f_{1,r} \sqsubseteq f_{2,r}$, $r = 1, \ldots, m_1$, then the underlying recursive procedure is called *locally time-monotonical.*

Lemma 6. *If a recursive procedure p is locally time-monotonical, then $f_1 \sqsubseteq f_2$ implies $\mathcal{T}(f_1) \leq \mathcal{T}(f_2)$.* □

Thus if we can prove that a certain recursive procedure p is locally time monotonical, then the timing behavior is monotonically distributed too.

For complex applications our constraints can be modified by applying *parameter space morphisms* (for details compare [BL94]). Such a morphism allows for concentrating on those parts of the parameter space that are essential for deriving space and time estimates.

As a practical application we present *balanced trees* which are interesting since operations defined upon them can easily be implemented by recursion and their recursion depth is usually bounded above by $O(\mathrm{ld}\ n)$, where n denotes the number of nodes in the tree. We study BB[α]-trees (cf. [Meh84]) in some detail. In Fig. 3 part of the specification of a BB[α]-tree package is given. Figure 4 shows all additional functions to be given by the programmer for implementing the recursive procedure `insert`. By applying the shown morphism, timing properties can be deduced involving the current number of nodes in the tree.

More details on real-time recursions can be found in [BL95] and in [BL94].

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

In order to study timing analysis of real-time objects, we discriminate between *the view from inside* and *the view from outside.*

The *worst-case performance (WCP)* of the object is estimated by help of a WCP-tool which facilitates the timing analysis. For example compare [HS91].

In order to ease the task of the WCP-tool, 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 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

```
generic

   size: natural;
   alpha: float;
   type element is private;
   with function "<"(left,right:element) return boolean is <>;

package BB_alpha_tree is

   type tree is limited private;

   procedure insert(an: element; into: tree);

   -- other operations suppressed

private
   type tree is
     record
       current_size: natural;      -- the current number of nodes in the tree
       -- other stuff representing the tree structure suppressed
     end record;
end BB_alpha_tree;
```

Fig. 3. Ada Code of Specification of BB[α]-tree (Fragment)

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. Of course, separating specification and implementation of timing information greatly improves modularization and facilitates testing of the real-time system.

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 simple parameters. The value of such a *state parameter* may change if the internal state of $\mathcal{O}$ changes due to 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).

In the following we investigate how gWCP and aWCP estimates can be added to Ada95.

4.1 Generic Worst-Case Performance Estimates

In order to augment Ada95 with gWCP estimates, we can exploit generic packa-

```
package body BB_alpha_tree is

  subtype node_number is natural range 0 .. size;

  recursive procedure insert(an: element; into: tree)

    with function morphism(t: tree)
        return node_number is
    begin
      return t.current_size;
    end morphism;

    with function recdep(current_size: node_number)
        return natural is
    begin
      return floor(1.0+(ld(current_size+1)-1.0)/ld(1.0/(1.0-alpha)));
    end recdep;

  is
  begin
    -- recursive implementation of insert
  end insert;
end BB_alpha_tree;
```

Fig. 4. Recursive Implementation of BB[α]-tree (Fragment)

ges and generic subprograms already present in Ada95. This feature even provides us with a generic parameter mechanism.

4.2 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. 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.
   This approach works well for type managers.
3. 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 (3a) suffers from the fact that the actual (discrete) type of the state parameter is not known in advance.

Approach (3b) 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.
- Declaring a *controlled* real-time type poses some problems:
  - A technique described in [Rat95] 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.

  These disadvantages give reasons why the techniques (1) and (2) are useful, and they justify the proposed language extensions.

Nevertheless Approach (3b) works well if tagged and limited controlled types and polymorphism are no problem for the real-time application.

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. Note that in case (1) and (2) this can be guaranteed by the language, while in case (3) the programmer is responsible for it.

As already mentioned, we are implementing a precompiler which translates Ada95 code augmented with our language extensions to pure Ada95 code. Out discriminants are mapped to ordinary record components which are enveloped in a protected type. In addition, a primitive operation returning the enveloped record component is 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.

Similarily, our precompiler envelops generic out parameters in protected objects just as it does for out discriminants. The generic out parameter itself is 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.

4.3 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.

As an example we study again BB[α]-trees. In particular 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 5 shows a specification of a corresponding Ada95 package. For didactic purposes we have assumed rather simple gWCP and aWCP estimates. 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.

```
generic

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

package Balanced_Tree is

  item_already_present: exception;

  procedure Insert(item: item_type)
      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;
```

Fig. 5. Specification of Balanced Tree Insertion

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

More details can be found in a forthcoming paper ([Bli95]).

5 Building a Precompiler based on GNAT

Although it has ever been seen as an integral part of Project WOOP[1] to implement a tool that incorporates the theoretical results gathered, it would have been a bold proposition to build a timing analysis tool and a compiler for extended Ada95 from scratch. Therefore and due to limited time and resources it was the availability of GNAT[2] that saved us a lot of work. To be specific, it prevented us from having to code a complete front-end for the whole Ada programming language while providing every facility we needed to build upon in order to incorporate *discrete loops*, *real-time recursions* and *real-time objects* into Ada95.

To further ease the task of implementation we decided to build a precompiler at first. This precompiler would have to translate our constructs to standard Ada. Any Ada compiler could then be used for translating the resulting code.

5.1 Extending the GNAT System

GNAT itself is a front-end and runtime system that uses the back-end of GCC as a retargetable code generator. The front-end uses an Abstract Syntax Tree as the underlying data structure and it comprises three phases[3] :

- Syntactic Analysis
- Semantic Analysis
- Transformation of the AST to a representation suitable for the back-end

Since only a minor part of the third phase is coded in a language other than Ada, the strengths of the Ada programming language also come into play in the source code of GNAT which properly reflects the structure of the corresponding chapters of the Reference Manual [Ada95]. This leads to a very modular functional design with no coupling between unrelated units.

Despite the fact that the current release of GNAT has already been validated, development of the compiler is not yet finished. This means that modifications of the current release of GNAT might have to be taken over to a future release. Therefore we have attempted to keep modifications of the original code as small

[1] WOOP, as an acronym for *Worst Case Performance of Object- Oriented Programs,* is a research project at the Department of Automation that is aimed at the determination of the timing-behavior of software for real-time systems.

[2] GNAT has been developed at New York University and its source code is distributed under terms of the GNU General Public License as published by the Free Software Foundation.

[3] Details can be found in [SB94]

as possible while providing 'extra'- functionality in separate units that are called where appropriate (e.g. on encountering keyword '*discrete*' during syntactic or semantic analysis). Because of arising dependencies it was on the other hand necessary to properly integrate our extensions into the original code.

Although these requirements seem to contradict each other, the hierarchical library mechanism of Ada95 actually made it possible to fulfill both of them in most cases.

5.2 A Case-Study on the Implementation of Discrete Loops

In this section we discuss the implementation of WPP—the WOOP Pre-Processor—with help of GNAT. Due to space considerations we picked the discrete loop construct to serve as an example.

The Abstract Syntax Tree (or AST for short) is perhaps GNAT's single most important data structure. It is constructed by the recursive descent parser and represents the input-program in a tree-like form. Subsequent processing in the front-end traverses the tree, transforming it in various ways and adding semantic information. Therefore no separate symbol table structure is needed. Every single piece of the Ada programming language finds its reflection in the AST. It has got nodes for statements, expressions, declarations, tasks and so on. We chose to treat discrete loops as a special form of for-loops and augmented the node N_Loop_Parameter_Specification (a descendant of N_Iteration_Scheme which is a descendant of N_Loop_Statement) with nodes for the *initial_value* and the *list_of_iteration_functions*.

AST-access is governed by a high-level interface that checks for the validity of the required access (you cannot convince GNAT to provide you with a loop-body out of a case-statement, for example). Since, for obvious reasons, GNAT is very pedantic about this, we created our own high-level interface for WOOP-specific parts of the AST. A tool that checks for the consistency of such modifications is provided with GNAT.

Modifying GNAT's Scanner and Parser was a straight-forward task since GNAT is very well structured. All we had to do was to make GNAT call our own function which would then parse the complete loop, put together the results in a subtree of the AST and return it to GNAT.

Semantic Analysis in general performs name and type resolution, decorates the AST with various attributes and performs all static legality checks on the program. Type resolution is done using a two-pass algorithm. During the first (bottom-up) pass, each node within a complete context is labeled with its type, or if overloaded with the set of possible meanings of each overloaded reference. During the second (top-down) pass, the type of the complete context is used to resolve ambiguities and to choose a unique meaning for each identifier in an overloaded expression [SB94]. In the case of a loop statement, GNAT has to analyze the loop's *iteration_scheme* and the body of the loop. Since the body of a discrete loop does not semantically differ from any other loop, we leave the latter task to GNAT. This of course does not prevent us from performing special checks on the body after semantic analysis has been completed.

The sole purpose of the analysis of the *iteration_scheme* of a discrete loop is to determine the type of the loop variable and to verify that this type is a *discrete* type. Please note that such an *iteration_scheme* contains three entities that provide type-information:

- The expression providing the *initial_value* for the loop variable.
- The *discrete_range*.
- The iteration functions contained in the *list_of_iteration_functions*.

Estimating the Number of Iterations of a Discrete Loop: Simple cases are solved by WPP itself. If things turn out to be more complex (e.g. complicated recurrence relations are involved), WPP passes this task over to Mathematica[4], a commercial package for computer algebra.

Code Generation: It is one of GNAT's built-in abilities to dump the source code from the generated tree. Although this feature was only meant for debugging purposes, it can be used for code generation too: as long as we are dealing with plain Ada95 code, GNAT can do all the work. On encountering a discrete loop, we take over to perform all transformations that are necessary in order to convert the loop to its equivalent in Ada95. The resulting code is then printed and control returned to GNAT.

More details concerning the implementation of WPP can be found in [BB95].

6 Conclusion

We have shown that by augmenting Ada with additional real-time language features, the language can be improved. In detail, we have introduced a new kind of loops, so-called *discrete loops*, we have shown that recursion can be used for real-time applications without harm, if a few conditions are met, and we have presented an approach how the interface of a class can be augmented by information concerning the worst-case time and space behavior.

Some of these new features require special care when implemented by a compiler. Although runtime checks can be performed in every case, the more compile time checks a compiler performs, the less runtime checks are necessary. Thus an "intelligent" compiler can save a lot of execution time.

[4] Mathematica is a registered trademark of Wolfram Research Inc.

References

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

[BB95] Johann Blieberger and Bernd Burgstaller. The role of GNAT within project WOOP. GNAT Workshop, Ada-Europe'95, 1995.

[BL94] Johann Blieberger and Roland Lieger. Worst-case space and time complexity of recursive procedures. *Real-Time Systems*, 1994. (to appear).

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

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

[Bli95] Johann Blieberger. Timing analysis of object-oriented real-time programs. (submitted), 1995.

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

[Meh84] Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, Berlin, 1984.

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

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

[Rat95] Intermetrics, Inc. *Ada95 Rationale*, 1995.

[SB94] E. Schonberg and B. Banner. The GNAT project: A GNU-Ada9X compiler. In *Conference Proceedings of TRI-Ada 94*, 1994.