TIMING ANALYSIS OF OBJECT-ORIENTED REAL-TIME PROGRAMS

JOHANN BLIEBERGER

December 20, 1995

Abstract. We investigate how information necessary for estimating the worst-case performance of methods of classes or objects, can be provided in an object-oriented way suitable for real-time programs.

This is done by taking into account class and object invariant information on the timing behavior. The latter one is expressed by generic parameters. In addition, the actual state of objects can also be used for determining their worst-case performance.

Furthermore, problems of performance estimates in heterogeneous networks and the relation between performance estimates and inheritance are discussed.

1. Introduction

Generally speaking, the interest of this paper 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 [Gwi92] 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 [MS95] 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 to be real-time, this holds for real-time systems too.

It is the purpose of this paper to show that abstract data types supplied with appropriate timing information are an excellent means to analyze the timing behavior of real-time programs.

Key words and phrases. object-orientation, programming languages, real-time systems, worst-case performance.

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

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++ ([ITM90]), only support *active real-time objects*. Other so-called object-oriented real-time programming languages, like Ada ([Ada95]), do not even support *real-time objects*.

Our proposed orthogonalism[1] 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 (cf. e.g. [LL73, SM92, Mok84, XP93]). In this paper, 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 [HS91]. Often the task of the tool is eased by restricting programming language features. For example by forbidding while-loops and recursion. In contrast, we promote the use of discrete loops (cf. [Bli94]) and real-time recursion (cf. [BL95, BL94]).

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.

[1] Compare [Weg87] for an excellent overview of orthogonal concepts in object-oriented language design.

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

3. Timing Information

In this section 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 Section 7.

Performance characteristics of algorithms and data structures have been studied extensively. For example compare [Knu73a, Knu81, Knu73b, Meh84, VF90, AHU74, AHU83, Sed88]. Commonly, this analysis is performed by informally introducing some sort of *complexity measure* (cf. [VF90]) or *size* (cf. [Meh84, AHU74]) of the data structure.

Our research on real-time loops and real-time recursions (cf. [Bli94, BL95, BL94]) has led us to the following conjecture.

Thesis 3.1. The relevant "parameters" of algorithms and data structures can be expressed in terms of *natural numbers*.

Foundation. There are two problematic issues in determining the timing behavior of algorithms w.r.t. real-time applications: *loops* and *recursions*. We investigate them in the following.

Loops: Computing the number of iterations for a *for-loop* is trivial.
Analyzing general loops is a much more difficult task. In order to avoid the problems connected with estimating the worst-case performance of general loops some researchers simply *forbid* general loops and force the programmer into supplying a constant upper bound for the number of iterations. Thus actually transforming the general loop into a for-loop with an additional exit-statement, or they directly require a constant time bound within which the loop has to complete (e.g. [PK89]). Other researchers attempt to do static and dynamic program path analysis using regular expressions (e.g. [Par93]).
We propagate the use of *discrete loops* for implementing real-time applications. The advantages of discrete loops have been presented in [Bli94]. We only repeat here that they are well-suited for implementing many important algorithms (e.g. all divide & conquer algorithms), but allow for efficient analysis of their number of iterations at compile time.

Anyway, we conclude that the number of iterations of a loop statement can be expressed by a natural number.

Recursion: Recursions in the context of real-time applications have been studied extensively in [BL95, BL94]. It has turned out that recursion can be used for implementing real-time applications without harm if a few conditions are met.

Time and space analysis of recursions is facilitated by use of parameter space morphisms[2]. In most cases it is advantageous if the target of this morphism are natural numbers. The reason for this is that a parameter space morphism is only useful if it simplifies the analysis of recursive procedures. In the following we discuss both time and space behavior.

Space: Stack space, even if it depends on the parameters of the recursive procedure, is measured in multiples of bytes. Thus stack space is a function from the set of tuples of parameters to the natural numbers.

Time: In case of timing analysis the only complicated statements are loops. To be more specific the number of loop iterations is relevant for the timing behavior of a recursive procedure, which again is a natural number. Thus the core of the timing analysis of a recursive procedure is a function from the set of tuples of parameters to the natural numbers.

Hence we can expect a useful parameter space morphism only if it maps the set of tuples of parameters to natural numbers.

More on this topic can be found in [BL94].

The above arguments show that natural numbers play a crucial role in investigating the time and space behavior of algorithms and hence support our thesis. □

Remark 3.1. Notice that Thesis 3.1 does not mean that timing information is expressed in simple terms of natural numbers. In fact rather complex expressions like

$$c_1 N \log N + c_2 N + c_3 \log N + c_4$$

are possible. It is however important that N denotes a natural number that enters the formula as a parameter and that c_1, c_2, c_3, c_4 are constants that on one side depend on the algorithm and data structure used and on the other side are machine dependent. More details can be found in Section 7.

Remark 3.2. A large number of articles and books have been published w.r.t. analysis of algorithms and data structures. In most cases a *complexity measure* (cf. [VF90]) or a *size* (cf. [Meh84, AHU74]) plays a central role. The arguments in the foundation of Thesis 3.1 provide reasons why natural numbers are that important in the analysis of algorithms and data structures.

4. WCP Estimates

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

[2] Details can be found in [BL94].

4.1. Definitions and Theoretical Results.

Definition 4.1. 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 4.2. 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 4.3. 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 $\mathcal{F}$, the current internal state of $\mathcal{O}$ and the implementation details of $\mathcal{M}_i$ to $\mathcal{D}$

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

Remark 4.1. Obviously, if WCP does not depend on $\mathcal{F}$ 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 paper we are going to investigate the general case of Definition 4.3.

Remark 4.2. 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 4.4. A *Worst-Case Space* 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 $\mathcal{F}$, the current internal state of $\mathcal{O}$ and the implementation details of $\mathcal{M}_i$ to a certain number of bits or bytes

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

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 4.3. In this context compare [Mey88], 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 4.5. 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 4.6. 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 $\mathfrak{p} = (p_1, \ldots, p_m)$, $p_1 \in \mathcal{P}_1, \ldots, p_m \in \mathcal{P}_m$ denote a vector of values of state parameters. Let $\mathfrak{P}$ denote the set of all such vectors.

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

Remark 4.4. Mentioning Thesis 3.1, we can suspect that both generic and state parameters are natural numbers. Thus deriving the required ordering relations should be straight-forward.

Definition 4.7. Let $\mathfrak{p}_1, \mathfrak{p}_2 \in \mathfrak{P}$. If $\mathfrak{p}_1 \leq \mathfrak{p}_2$ implies that $\mathsf{aWCP}(\mathfrak{p}_1) \leq \mathsf{aWCP}(\mathfrak{p}_2)$ 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 4.1. Worst-case performance estimates of time-monotonical objects can be ordered according to the following hierarchy

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

Proof. To prove the inequality, we have to consider two cases:

(1) There exists an upper bound for all state parameters.
Obviously generic parameters cannot do better than these upper bounds.
(2) There does not exist an upper bound for all state parameters.
Thus it is not possible to find a valid gWCP estimate better than ∞.

□

Remark 4.5. Theorem 4.1 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.

4.2. 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 (cf. [Ada95]) 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 both 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.
This is difficult because either
- we have to fix a certain name for this operation or
- the programmer has to extend a predefined abstract type called say `real-time`, which provides this operation.

We consider the first approach ugly. The second one artificially increases the number of tagged types. Moreover, 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.

(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 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.

4.3. Example. As an example we study a balanced tree structure such as a 2-3-tree, an AVL-tree, or a BB[α]-tree (cf. [Meh84]). 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 1 shows a specification of a corresponding Ada95 package. For didactic purposes we have assumed rather simple gWCP and aWCP estimates, which for example are not correct in case the $R = 0$. 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.

```
generic

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

package Balanced_Tree is

  type tree(R: out natural := 0) is private;

  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 1. Specification of Balanced Tree Insertion

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 (cf. e.g. [GKP89])

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

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 functions if the corresponding bodies (implementation parts) are compiled. In our example, if a discrete loop (cf. [Bli94]) 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.

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 in [Ada95].

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 [Ada95] 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 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 function 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[3].

Before the code of a method M is executed, the expressions of the corresponding state parameter change 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 6.

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.
- 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. 2. Then one can define a generic package `g` shown in Fig. 3. In order to enable this syntax, we need two attributes. Assume the predefined types depicted in Fig. 4. 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`.

5. Semantics of Out Discriminants and Generic Out Parameters

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

- type manager and

[3]Again refer to [Mey88] for how semantics can be defined in an object-oriented framework.

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

procedure o(v: t)
  with duration <= ...;
```

Figure 2. A tagged type and a primitive operation

```
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 3. 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 4. Some predefined types

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

  protected type protected_discrete is
    entry Lock;
    entry Set(to: in discrete);
    entry Read(val: out discrete);
    entry Value(val: out discrete);
    entry Unlock;
  private
    locked: boolean := false;
    val: discrete;
  end protected_discrete;

end Protected_Types;
```

FIGURE 5. Specification of Package `Protected_Types`

- 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 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 parameter, have to be supplied with initial values.

5.1. Out Discriminants. Currently, we are implementing a precompiler which translates Ada95 code augmented with the syntax and semantics described in Section 4.2 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 an entry call of 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 (cf. Figures 5 and 6). The given operations are supposed to be used for setting and retrieving the value of the underlying out discriminant. Entry `Read` is supposed for retrieving the value of the protected type from outside, `Value` is used for retrieving the value within the implementation of the primitive operation, i.e., during the out discriminant is locked. The precompiler inserts a call to `Lock` before the statements of the body of a primitive operation and a call to `Unlock` at the end of the statements, i.e., before each return statement and before the end statement.

The reasons why state parameters can be supposed to be of discrete type or even of type natural, are given in Section 3.

```
package body Protected_Types is

  protected body protected_discrete is

    entry Lock when not locked is
    begin
      locked := true;
    end Lock;

    entry Set(to: in discrete) when locked is
    begin
      val := to;
    end set;

    entry Read(val: out discrete) when not locked is
    begin
      val := val;
    end read;

    entry Value(val: out discrete) when locked is
    begin
      val := val;
    end value;

    entry Unlock when locked is
    begin
      locked := false;
    end Unlock;

  end protected_discrete;

end Protected_Types;
```

Figure 6. Implementation of Package `Protected_Types`

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 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.

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 to this function which internally is forwarded to an entry call of the protected object.

6. Runtime and Compile Time Checks

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

6.1. 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[4] and "inserts" this estimate at the beginning of the basic block into the machine code.
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.

ACT: *The specified duration is compared to the actual execution time of the subprogram.*

[4] See [HS91] for the notion of basic blocks.

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 a **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 6.1. We have assumed that the scheduler does not preempt a basic block. Dispatching points are only at the beginning of a basic block.

6.2. 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.
(2) Investigate recursive subprograms if present to ensure that they meet certain requirements (cf. e.g. [BL94]).
(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/checks. In fact, the more the compiler can prove, the less runtime checks are necessary.

Within Project WOOP we rely on discrete loops to perform (1) (cf. [Bli94]) and on real-time recursions to check (2) (cf. [BL94]). Control and data flow analysis (3) is done with help of Mathematica[5], a computer program for doing symbolic manipulations of mathematical expressions. In addition, Mathematica is especially useful for solving recurrence relations that occur in the context of discrete loops and real-time recursions. However, it can also be used for control and data flow analysis by translating part of the imperative language Ada to Mathematica's internal Lisp-like, functional language. At last, Mathematica is used for deciding whether the expression describing the timing behavior of the body of a constraint subprogram is smaller than or equal to the WCP estimate provided at the interface of the subprogram.

7. Machine-Dependent Performance Characteristics

Implementing a distributed real-time application for a heterogeneous network using *one* programming language for that purpose, poses interesting problems for how to specify timing information at the objects' interfaces. In this section we present a solution to these problems by using the object-oriented features of Ada95.

We propose to use abstract functions for time and space specifications and one child package for each type of CPU in the network. Such a child package implements

[5]Mathematica is registered trademark of Wolfram Research Inc.

the appropriate time and space specifications. The following is a simple example illustrating these ideas.

A generic package `WCP` is shown in Figure 7. The abstract function `M_Time` is sup-

```
generic
  N: natural; -- generic parameter
package WCP is

  type t is private;

  function M_Time return duration is abstract;
    -- abstract function describing the timing
    -- behavior of procedure M

  procedure M(v: t)
    with duration <= M_Time;

private

  ...

end WCP;
```

Figure 7. Abstract Specification of Timing Behavior

posed to compute the timing behavior of procedure `M`. Procedure `M` is implemented by standard means in the body of package `WCP` and is not shown here.

For each type of CPU in the network that runs Package `WCP`, a child package of `WCP` has to be created. For example see Package `WCP.CPU` in Figure 8. The

```
generic
package WCP.CPU is
  function M_Time return duration;
    -- non-abstract declaration of M_Time
end WCP.CPU;
```

Figure 8. CPU-specific Specification of Timing Behavior

corresponding package body is depicted in Figure 9. Note that it is not necessary to reimplement Procedure `M`, only the CPU-specific time behavior has to be given. Both packages can be instantiated by standard means (see Figures 10 and 11).

After successfully compiling these packages, they can be assigned to partitions according to the type of CPU (cf. [Ada95], Annex E). The linker can check whether the assignment is appropriate.

This approach does not distinguish between the three parts worst-case estimates consist of, namely

(1) generic or actual state parameters
(2) algorithmic parts (e.g. number of iterations, recursion depth of a recursive procedure or function, ...)
(3) machine dependent parts (e.g. performance of CPU, ...)

Actually only part (3) has to be changed if a package is to be run on a new CPU, but the object-oriented approach does not provide an adequate solution, which is fine-grained enough for this purpose.

```
with Ada.Numerics.Generic_Elementary_Functions;

package body WCP.CPU is

  type my_float is digits 10;

  package My_Numerics is new
    Ada.Numerics.Generic_Elementary_Functions(
      Float_Type => my_float);

  use My_Numerics;

  function M_Time return duration is
    -- actual implementation for CPU
  begin
    return duration(1.23*my_float(N)*log(my_float(N))+
      13.5*my_float(N)-1.45*log((my_float(N))+24.78));
  end M_Time;

end WCP.CPU;
```

Figure 9. CPU-specific Implementation of Timing Behavior

```
with WCP;
package my_WCP is new WCP(N => 100);
```

Figure 10. Instantiating a `WCP`-package

```
with WCP, WCP.CPU;
package my_WCP.my_CPU is new my_WCP.CPU;
```

Figure 11. Instantiating a `WCP.CPU`-package

An alternate solution would provide new language features for this purpose. This, however, would increase language complexity. Thus we chose the approach presented in this section for the implementation of our precompiler.

8. Inheritance and WCP-Estimates

Inheritance is the essential property of object-oriented programming. In general, inheritance allows for extending the representation of a type

(1) by extending the number of the instance variables and
(2) by overriding the implementation of certain methods.

The open question in our context is how timing and space estimates are affected by inheritance.

It is however clear that if a certain method is overridden by a new implementation, then the timing and space behavior of that method can be suspected to have changed. Thus it is obvious that we allow time and space estimates to be overridden during inheritance.

Note, however, that this makes the task of a WCP-tool, like that described in [HS91], more difficult, since timing estimates in conjunction with polymorphism either implies overestimating or incorporating more knowledge on the implemented algorithm. For example this problem might be solved by applying an approach similar to that described in [Par93].

9. Conclusion

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 paper, 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. The development of such a compiler is part of Project WOOP 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 helpful discussions. Special thanks go to Bernhard Scholz, Bernd Burgstaller, and Roland Lieger.

References

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

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[BL94] Johann Blieberger and Roland Lieger. Worst-case space and time complexity of recursive procedures. (submitted), 1994.

[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.

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

[Gwi92] J.M. Gwinn. Object-oriented programs in realtime. *ACM SIGPLAN Notices*, 27(2):47–56, 1992.

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

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

[Knu73a] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., second edition, 1973.

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

[Knu81] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., second edition, 1981.

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

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

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, NY, 1988.

[Mok84] Aloysius 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.

[MS95] Stuart Maclean and Sean Smith. Object-oriented programming for embedded systems. *ACM SIGPLAN Notices*, 30(9):3–8, 1995.

[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.

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

[SM92] Alexander D. Stoyenko and Thomas J. Marlowe. Polynomial-time transformations and schedulability analysis of parallel real-time programs with restricted resource contention. *The Journal of Real-Time Systems*, 4:307–329, 1992.

[VF90] Jeffrey S. Vitter and Phillipe Flajolet. Average-case analysis of algorithms and data structures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 431–524. North-Holland, 1990.

[Weg87] Peter Wegner. Dimensions of object-based language design. In *OOPSLA Proceedings*, pages 168–182, 1987.

[XP93] Jia Xu and David Lorge Parnas. On satisfying timing constraints in hard real-time systems. *IEEE Trans. Software Engineering*, 19(1):70–84, 1993.

DEPARTMENT OF AUTOMATION (183/1), TECHNICAL UNIVERSITY OF VIENNA, TREITLSTR. 1/4, 1040 VIENNA, AUSTRIA

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