The Deadline Command

Colin Fidge[1] *Ian Hayes*[1,2] *Geoffrey Watson*[1]
[1]Software Verification Research Centre
[2]Department of Computer Science and Electrical Engineering
The University of Queensland
Queensland 4072, Australia

Abstract

The deadline command is a simple, yet highly flexible, language primitive for real-time programming. Formally, it has a straightforward definition as a constraint on the current time. In practice, it acts as a compiler directive, which can be checked statically, to ensure that the program will always meet its timing requirements. As a concrete example, a way of adding the deadline command to the safety-critical SPARK programming language is described. A detailed comparison is made with other proposed real-time language constructs.

1 Introduction

Programming real-time systems using a high-level language is notoriously difficult because the ultimate timing behaviour of the program depends on the compilation strategy and target architecture. The real-time behaviour of the executable code is not visible in the high-level language source program.

Unfortunately, contemporary programming languages, even those promoted for use in real-time applications, provide few constructs for directly expressing timing properties. Consequently, real-time programmers resort to testing or inspection of compiled machine code to check timing correctness, or may avoid high-level languages altogether, thus losing the productivity benefits afforded by such languages.

Here we address this problem by introducing a small set of programming language constructs for stating enforceable timing requirements. The principal feature is a 'deadline' command [1] for expressing upper timing bounds, complementing existing delay statements which express lower timing bounds. The deadline command

- is a simple, rather than compound, statement and can thus express fine-grained timing requirements, including those that do not necessarily follow the block structure of the program,
- is flexible enough to be applicable to any imperative programming language,
- has a straightforward semantic definition, and
- can serve as a statically-analysable compiler directive for a priori guarantees of run-time timing correctness.

Here we give an overview of this language construct and its implementation for sequential program fragments, propose an instantiation as an extension to the high-integrity SPARK programming language, and compare it with related work.

2 Previous work

Although many programming languages are claimed to be suitable for 'real-time' programming, few offer language constructs for *directly* expressing the desired timing behaviour of the executable code. At best, contemporary languages offer a handful of statements for accessing the hardware clock. They leave proof that the program obeys its timing constraints to run-time testing or machine code analyses undertaken outside the language itself.

Widely-used languages that currently offer some real-time support include C, occam 2 and, in particular, Ada 95 [2]. The major features they provide are as follows.

Time type In order to manipulate times in a program we need a suitable type for declaring time-valued variables and constants. Let imaginary type 'time' denote the 'real-world' time frame. In practice, a programming language will offer a type whose values give an acceptable approximation to this type. For instance, Ada 95 has a `Real_Time` package that provides a type `Time` whose progress approximates the passage of absolute real time, and a type `Time_Span` for representing relative durations. Thus, in the scope of the `Real_Time` package, we can declare a variable t as one intended to hold a time as follows.

$$t : \texttt{Time};$$

Telling the time Another common feature is a statement for reading the (hardware clock) time into a local variable. For instance, Ada 95's `Real_Time` package provides a `Clock` function that can be used to read the current absolute time into a variable t.

$$t := \texttt{Clock};$$

To define the meaning of this statement, let imaginary variable now_0, of type time, denote the moment at which this statement begins, and now denote the moment at which it ends. The above statement can then be formally defined as setting t equal to some time between the starting and finishing times of the statement (assuming the clock's resolution is sufficiently fine). In other words, it makes the following property true [3].

$$\mathsf{now}_0 \leqslant t \leqslant \mathsf{now}$$

Delaying An important feature is a delay statement for halting progress until a significant time. For instance, Ada 95 has a **delay until** statement which accepts a `Time`-valued expression as an argument. It can be used as follows to ensure that it is no earlier than time $t+10$ when the statement finishes.

$$\textbf{delay until } t + 10.0;$$

Formally, this statement can be defined in terms of our imaginary current time variable as one that makes the following property true [3].

$$t + 10 \leqslant \mathsf{now}$$

The only other explicit real-time language construct normally provided is some form of timeout facility. Either the program can timeout while awaiting a communication event, or while awaiting completion of some local computation. In effect, a timeout is an error-recovery mechanism, allowing the program to continue when an anticipated event fails to occur. The timing requirements expressed by timeout statements are thus *soft* because failure to meet them is a recoverable situation. We will not consider timeouts further—our interest here is with *hard* timing requirements only, i.e., those that must be proven never to fail.

Other support for real-time programming tends to be implicit and implementation specific. Importantly, Ada 95 offers a major advance over other languages by requiring developers of real-time Ada compilers to clearly document implementation-specific features that influence our ability to reason about a program's run-time behaviour [2, §D]. For example, the worst-case rate of clock drift between Ada's type `Time` and 'real' time must be known, and the type's range must span at least fifty years. Similarly, the performance of the clock function and delay statement must be clearly defined so that we know, for instance, by how much a delay may overrun its stated time, or what the effect will be of attempting to delay until a time that has already passed.

Nevertheless, the burden of applying this knowledge to ensure that programs satisfy their real-time requirements still rests with the programmer, rather than with the programming environment. Although Ada 95 is one of the most advanced real-time programming languages currently available, it still forces programmers to achieve their desired timing requirements by first implementing the functional behaviour using imperative language constructs, and then undertaking separate timing analyses and tests, rather than stating the timing requirements directly in the program, and leaving it to the programming environment to achieve them.

3 Real-time programming constructs

Our goal, therefore, is to introduce real-time programming constructs that allow timing constraints to be directly expressed in a natural way, and automatically enforced at compile time.

As a motivating example, consider that a common real-time requirement is to perform some action periodically. After some starting time S, an action must be invoked repeatedly, with period T. Each invocation must finish within a deadline D, relative to the start of the period. Often the behaviour is required only while the system is in a particular mode, defined here by boolean expression M.

In current programming languages this is typically expressed as shown in Figure 1. The period constant T appears explicitly, and is used in the delay statement to ensure that the loop does not iterate too quickly. Similarly, the first starting time S is used to determine when the action may be invoked for the first time. However, the requirement that the action executes quickly enough, as defined by the desired deadline constant D, does not appear. It is up to the programmer to establish this, either by testing or some sort of separate timing analysis. Furthermore, there is no way to know whether time S has already passed or not when the program fragment begins. If it has, then it may be impossible to meet the first deadline. Again, current languages offer no way of even stating this timing assumption.

To solve these deficiencies we propose the introduction of a small set of generic timing constructs to the programming language.

Assertions Assertions are a well-established way of specifying a property that is expected to be true at some point in a program. They typically appear as an expression in braces which, while not executable, serves to document assumptions, or aid program analysis. Here, we extend the concept by allowing the imaginary variable now to appear in assertions to represent the time at which this point in the program is reached. For instance, we can state that that current time is expected to be no later than 25 as follows.

$$\{\mathsf{now} \leqslant 25\};$$

Enforcing deadlines The principal feature of our approach is the deadline command. Like an assertion, it is not executable—its use and implementation are discussed further below. It accepts an absolute time-valued expression and requires the current time to be no later than this value when the statement is reached. For instance, we can state that this point in the code must be reached by time $t + 13$ as follows.

$$\mathsf{deadline}\ t + 13;$$

Formally, this statement is defined as one that makes the following expression true [3].

$$\mathsf{now} \leqslant t + 13$$

In particular, note the symmetry with the delay statement in Section 2.

Logical constants In Figure 1 we saw examples of compile-time constants being declared for use in executable statements. Having introduced two non-executable constructs, we also allow *logical* constants (specification variables) to be declared for use within these constructs. A logical constant may appear only within assertions or deadline commands. This facility can be used to declare constants of 'real' type time (as opposed to the programming-language approximation Time). Formally, the value of a logical constant is chosen *angelically*,

```
declare
    S : constant Time := ...;              -- earliest starting time for first action
    T : constant Time_Span := ...;         -- period
    next : Time;                           -- earliest starting time for each action
begin
    next := S;
    while M loop
        delay until next;                  -- wait until starting time for this action
        -- Action
        next := next + T;                  -- calculate next action's starting time
    end loop;
end;
```

Figure 1: Periodic behaviour in Ada 95.

so once declared it may be constrained by a subsequent assertion. For example, we can declare a logical constant c, and then later use it to capture the current time.

$$
\begin{array}{c}
c : \mathsf{con\ time}; \\
\vdots \\
\{c = \mathsf{now}\};
\end{array}
$$

Returning to our motivational example, Figure 2 shows the same program annotated with the new timing constructs. The relative deadline D for each action now appears explicitly. The code block has an initial assertion stating that the current time, when this program fragment begins, is expected to be less than starting time S. This contextual information can be used when verifying whether the first invocation of the action can be completed in time or not. There is also a deadline command to constrain each action invocation. It states that each action must finish within D time units of the earliest time at which the action may be invoked. These constructs thus allow *all* the timing constraints to be unambiguously expressed.

4 Implementing the real-time constructs

We now consider how the new constructs can be implemented. Firstly, recall that an assertion merely documents assumed properties, and a logical constant is used in non-executable constructs only. From a compiler's perspective, therefore, assertions and logical constants in high-level language programs can be ignored, i.e., treated like comments. This is equally true for our time-specific assertions and constants.

The deadline command offers a much greater challenge, however. Whereas a delay statement can be implemented by generating code that slows the program's progress, a deadline that is in danger of being missed would appear to require the compiler to generate code that makes the program run *faster*. It is not surprising that no existing language implements such statements!

The solution to this dilemma is to recognise that a deadline command does not itself result in executable code, but is instead a *directive* requiring the compiler, or an associated tool, to guarantee that the executable code generated for the program will always meet the specified deadline. This can be done in three phases [3].

In the first phase, control-flow analysis is used to find all execution paths that end at a deadline command. In the example in Figure 2, there is one deadline command which cuts the loop into four distinct paths, as shown in Figure 3. Guard nodes '[B]' indicate that boolean condition B must be true for this path to be followed.

1. The *loop-entry* path goes from the beginning of the program fragment to the deadline command.

```
declare
    S : constant Time := ...;
    T : constant Time_Span := ...;
    D : constant Time_Span := ...;      -- relative deadline for each action
    next : Time;
begin
    {now < S};                          -- latest starting time for this code
    next := S;
    while M loop
        delay until next;
        -- Action
        deadline next + D;              -- latest finishing time for this action
        next := next + T;
    end loop;
end;
```

Figure 2: Periodic behaviour extended with timing constructs.

2. The *iteration* path goes from the deadline command back to itself, representing a *single* iteration of the loop.

3. The *loop-exit* path goes from the deadline command, via the false loop condition, to exit the program fragment.

4. The *bypass* path does not enter the loop at all, representing the case where the loop condition was initially false.

In this particular program fragment, paths 1 and 2 both end at a deadline command, so their timing must be checked.

In the second phase, a sufficient timing constraint is found for each such path. This is done using the deadline expression at the end of a path, and the statements in the path itself. For instance, consider path 1, which consists of the following sequence of statements.

1. $\{\mathsf{now} < S\}$

2. $next := S$

3. $[M]$

4. **delay until** $next$

5. Action

6. deadline $next + D$

(Statement 3 represents run-time evaluation of the loop expression to 'true'.) To determine the timing constraint on this path, we observe from the formal meaning of the deadline command that statement 6 requires us to satisfy expression '$\mathsf{now} \leqslant next + D$'. From statement 2 we know that $next$ equals S (assuming 'Action' does not change $next$), so the requirement can be re-expressed as '$\mathsf{now} \leqslant S + D$'. From statement 1 we know that this sequence of statements is intended to begin before time S. In this situation the difference between the worst-case starting time and the desired finishing time may be as small as $(S+D)-S = D$. A timing constraint on path 1 that will guarantee the deadline is met is therefore to always execute this sequence of statements in D seconds or less.

In the third phase, conventional worst-case execution time prediction techniques are applied to the machine code generated by the compiler for these paths, to prove that the code will run quickly enough. Note that each path identified above corresponds to a linear, non-branching sequence of machine instructions, and the timing constraint is a simple end-to-end worst-case execution time requirement. We therefore merely need to apply a 'cycle counting' procedure to the generated machine code. For RISC architectures, which involve pipelining and caching, sophisticated timing prediction techniques are

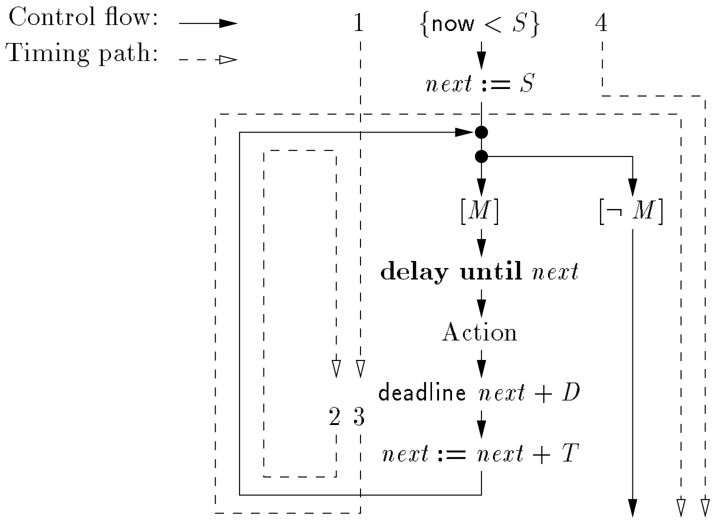

Figure 3: Flow of control for periodic program.

now available which offer a high degree of accuracy for linear paths [4]. The analysis is not entirely trivial, however. The delay statement in the middle of path 1 splits the analysis into two parts: one where the delay statement is reached before, and one where it is reached after, time *next* [3].

In practice, the three phases could be implemented in two distinct ways. Firstly, the compiler could undertake timing analysis as it parses the program and generates code. Indeed, standard compilation techniques build intermediate program representations that are ideally suited to such analysis. Alternatively, the code generated by the compiler could be analysed by a separate tool. This requires a way of matching corresponding points in the source program and object code [5], but has the advantage of leaving an existing software development environment unchanged. In either case, an inability to meet the timing constraints must result in appropriate error feedback to the programmer, who must then modify the program, or its environment, accordingly.

5 SPARK instantiation

SPARK is an extended subset of Ada intended for programming high-integrity or safety-critical systems [6]. The SPARK environment is a good starting point for instantiating our real-time language constructs because its philosophy already has many features in common with our approach.

- Non-executable *annotations* can appear in SPARK programs as special comments, prefixed by '--#'. They are ignored when SPARK programs are compiled by an Ada compiler, but are meaningful to separate SPARK analysis tools.
- Assertions about program variables and constants can appear in SPARK programs as annotations.

  --# **assert** $0 < x$;

- The SPARK Examiner tool undertakes information and data-flow analysis of SPARK programs to check for uninitialised variables, ineffective assignments and other coding errors.

However, SPARK presently offers no direct support for real-time programming [6, p.7]. Here we propose extensions to SPARK to embrace our timing concepts.

Unfortunately, the real-time Ada 95 constructs described in Section 2 above are not

```
procedure Main is
    Arrival : Spark_Time.Time;         -- starting time for each cycle
begin
    Arrival := Spark_Time.Clock;       -- read current time
    Arrival := Arrival + 0.005;        -- compute earliest time for first cycle
    while true loop
        --@ deadline Arrival;          -- latest finishing time for preceding cycle
        delay until Arrival;           -- earliest starting time for this cycle
        Control(Arrival);
        Arrival := Arrival + 0.02;     -- compute starting time for next cycle
    end loop;
end Main;
```

Figure 4: Time-annotated SPARK autopilot program.

available by default in SPARK. Nevertheless, Barnes shows that these timing constructs can be reintroduced by creating a SPARK package to act as an interface between the core SPARK language and the Ada 95 real-time features [6, §8.3]. This allows a time type, and clock read and delay statements to be made accessible in SPARK, while highlighting the fact that their implementation is not checked by the SPARK tools. For brevity here, we will assume this has been done, to create a package called `Spark_Time` which follows the Ada 95 conventions.

Our remaining constructs can be introduced by extending SPARK's annotation mechanism. We will use a distinct '`--@`' notation, so that existing SPARK tools will not try to examine our real-time annotations. The types used must all be known in the SPARK environment, so we always use the implementable type `Time` from the `Spark_Time` package here, rather than the logical type `time`.

- We allow 'timed' assertions which may use the special variable `Now` of type `Time`.

  --@ **assert** `Now` < 25.0;

- We introduce a new annotation for declaring logical constants, including `Time`-valued constants.

  --@ c : **con** `Spark_Time.Time`;

  (This is consistent with the existing SPARK annotation for declaring *proof functions*, i.e., functions that are introduced to assist with program verification, but do not generate object code [6, §11.3].)

- We introduce a new annotation for stating deadlines.

  --@ **deadline** $t + 13.0$;

  That this is an annotation, and not a statement, emphasises the non-executable nature of deadline commands. (Formally, it is consistent with the existing SPARK annotation for stating postconditions, i.e., conditions that must be true when a subroutine finishes [6, §3.4].)

These syntactic extensions would be sufficient to support the addition of a timing analyser to the SPARK toolkit, following the principles outlined in Section 4 above.

5.1 Time-annotated SPARK example: Autopilot

Here we give an example of our real-time annotations in a SPARK code fragment. It is based on the autopilot control system described by Barnes [6, §13.3] in which a set of navigational instruments is read periodically and the readings are used to update the aircraft's heading and altitude. In this version we have added timing constraints similar to those suggested by Stoyenko et al [7].

```
procedure Control(Started : in Spark_Time.Time) is
    --@ DoneReading : con Spark_Time.Time;
begin
        ⋮  -- read instruments
    --@ deadline Started + 0.005;          -- latest time to complete readings
    --@ assert DoneReading = Now;          -- time at which readings finished
        ⋮  -- update controls
    --@ deadline DoneReading + 0.003;      -- latest time to complete updates
end Control;
```

Figure 5: Time-annotated SPARK control procedure.

1. Control cycles must be performed every 20 ms after an initial latency of at least 5 ms.

2. All instrument and switch readings must be completed within the first 5 ms of each period.

3. All control updates must be completed within 3 ms of reading the instruments.

As shown in Figure 4, the first of these requirements is achieved by adding a variable *Arrival* to the infinite loop in the main program. (Non-real-time SPARK annotations are not shown. Items from our proposed `Spark_Time` package must be explicitly prefixed by the package name because SPARK prohibits Ada's **use** clause [6, p.154].) *Arrival* holds the absolute time at which each control cycle is meant to begin. It is initialised by reading the clock and advanced to allow for the initial latency. It is then advanced by the period at each iteration. A delay statement is used to enforce the periodic timing constraint, and a deadline command to ensure that each control cycle finishes in time for the next cycle to start. Placing the deadline at the beginning of the loop ensures that the first control cycle begins in a timely manner, as well as each subsequent cycle.

The other two timing constraints are on the time for reading the instruments and the time for updating the controls. As shown in Figure 5, these can both be expressed as deadlines in the *Control* procedure. The first deadline is relative to the start of each cycle as defined by the *Arrival* variable in the main program, which is passed as parameter *Started*. By marking this starting time *outside* the procedure, we include the run-time overheads of implementing the procedure call in the timing constraint. Lastly, logical constant *DoneReading* is introduced to mark the time at which the readings were completed so that the second deadline can be expressed relative to this moment.

6 Comparison with related work

The TimeC language is motivated by concerns similar to ours [8]. It is a language independent notation for expressing timing constraints that can be enforced at compile time. Firstly, significant points in the program are marked with meaningful comments. For instance, the starting and finishing points of some action A can be marked as follows.

```
/*! start */ {
A
} /*! end */
```

Then a separate timing requirement is expressed outside the program text. For instance, a requirement that A takes no more than D time units to execute would be expressed as follows.

$$\mathsf{time}(end) - \mathsf{time}(start) \leqslant D$$

In our notation, this requirement would be expressed entirely within the program text using a logical time-valued constant c to mark the time at which the start of the block is reached.

$$
\begin{array}{l}
\{c = \mathsf{now}\}; \\
A; \\
\mathsf{deadline}\ c + D;
\end{array}
$$

Separating the timing requirements from the program text creates a significant problem for TimeC, however. In conditional code some time markers may not be reached, raising the question of how to assess timing requirements referring to markers that were not passed. Special notations for expressing 'disjunctive' requirements are added to TimeC to solve this. Such notations are unnecessary in our case. By embedding the timing notations in the program, we guarantee that the timing requirements analysed are those that are relevant to the actual control path being studied.

Many proposals express timing constraints as attributes of program blocks. For instance, the Distributed Programming System (DPS) uses *temporal scopes* which associate constraints with sequences of program statements [9]. Timing constraints can be placed on the starting and finishing time of code blocks and may refer to relative or absolute times. For example, a requirement that a sequence of statements A starts no earlier than absolute time T, and finishes no later than absolute time U, is expressed in DPS as follows.

$$
\begin{array}{l}
\textbf{start at}\ T\ \textbf{by}\ U\ \textbf{do} \\
\quad A \\
\textbf{end}
\end{array}
$$

This effect can be similarly achieved in our notation from Section 3.

$$
\begin{array}{l}
\textbf{delay until}\ T; \\
A; \\
\mathsf{deadline}\ U;
\end{array}
$$

An important difference, however, is that the DPS timing constraints are attached to code blocks, whereas ours are primitive statements. This makes our notation more flexible. Our timing constraints apply to control flow paths, rather than the program's block structure, and are thus free to enter and exit structured statements such as loops, conditionals, subroutines, etc. DPS also differs in basic philosophy from our system in that its timing constraints are checked at run time and an exception raised if they are not met, although compile-time verification is mentioned as future work.

Similarly, the Maruti Programming Language also associates timing constraints with program blocks [10]. Both relative and absolute deadlines can be expressed, and act as scheduling requirements for the Maruti system's scheduling pre-processor. This requires that the execution time of each basic block is known in advance—the execution times used are estimates provided by the programmer.

The proposed CRL language [7], allows timing constraints to be associated with the start and end of any block or statement. CRL timing constraints are intended to be checked at compile time, in a static schedulability test. (CRL also has an '`assert`' construct which generates a *run time* check.) Stoyenko et al [7] present an aircraft navigation control system similar to that in Section 5.1 above. In CRL the periodic constraint in the main loop of the control thread is expressed as a requirement to cycle every 20 ms after an initial latency of 5 ms relative to the start of the thread.

```
thread control
  activationdeactivationconstraint
    periodic use (20)
    firstactive nosoonerthan (5)
  endactivationdeactivationconstraint
    ⋮  % control code
endthread control
```

The constraints on the time for reading the instruments and the time for updating the controls are expressed in two different ways in CRL. The first is by a constraint on the method that reads the sensors to finish within 5 ms of the start of the method.

```
method update_status
    timeconstraint
        nolaterthan (5)
    endtimeconstraint
        ⋮  % read instruments
endmethod update_status
```

The second constraint is defined within the control thread itself. This is done by enclosing the constrained code in a block. The point prior to the block is given a label *read_stat*, and the 3 ms timing constraint is then expressed relative to this.

```
thread control
        ⋮
    call self-update_status()
    !read_stat$
    block
        ⋮  % update controls
    endblock
    timeconstraint
        nolaterthanrelative
            (read_stat 3 local)
    endtimeconstraint
endthread control
```

CRL time labels can thus perform the same function as our logical time-valued constants.

The Timed-Constrained Event Language (TCEL) also seeks to enforce timing constraints at compile time [11]. Although TCEL associates its constraints with blocks of code, it makes a distinction between local and externally-observable actions within blocks. Timing constraints apply to observable events only. This potentially allows a compiler to use flow analysis to improve the program's timing behaviour by moving unobservable code segments between blocks. Gerber and Hong present an aircraft flight controller in TCEL [11], which can be made similar to that in Section 5.1 above.

```
every 20 ms finish within 8 ms
    do
    {
        /* read instruments */
    }
    finish within 3 ms
    {
        /* update controls */
    }
```

The first line defines a periodic behaviour, with period 20 ms and overall deadline of 8 ms from the start of each period. Each period is then partitioned into a *reference* block and a *constraint* block. The second of these has a distinct timing constraint consisting of a deadline of 3 ms, relative to the completion of the last *observable* statement in the first block. (This is not quite the same as our autopilot above, or the CRL version, because if it is the case that updating the controls always takes less than 3 ms, the TCEL program would potentially allow reading the instruments to take longer than 5 ms.)

Chung and Dietz propose a fine-grained timing notation for extending imperative languages as part of the Compiler for Hard Real-Time Systems (CHaRTS) method [12]. They assume that real time programs always consist of a set of periodic activities. Externally observable events are distinguished via special consideration of volatile variables. Block-structured language constructs are proposed to declare periodic tasks, label blocks, and express timing constraints on the starting and ending times of blocks. For instance, a relative timing constraint of D on block A is expressed by introducing a *time variable* label *tv* as follows.

```
@tv:{
    A;
}
@: .tv + D < tv.;
```

The prefixed and suffixed dots '.' denote the start and end times of the labelled statement respectively. We do not need such a notation because our timing constructs mark points *between* conventional program statements, rather than labelling the statements themselves. An

```
s : con seq time;             -- declare logical sequence of times
    ⋮
n := 0;
{s(0) = now};                 -- mark starting time
while n < m loop
    deadline s(n) + D;        -- latest starting time of (n + 1)st iteration
    A;
    n := n + 1;
    {s(n) = now};             -- mark finishing time of nth iteration
end loop;
```

Figure 6: Constraining loop iteration overheads.

unusual feature of the CHaRTS notation is an indexing mechanism for relating different iterations of loop statements. For instance, to express the notion that a loop statement must 'iterate', i.e., go from the end of the loop body back to its beginning, in no more than D time units, the following notation is used [12, Fig. 3].

```
n = 0;
@tv:{
    while(n < m) {
        A;
        n = n + 1;
        @: .tv[2] < tv.[1] + D;
}}
```

This states that the starting time of the second iteration must be less than the finishing time of the first iteration plus D. As shown in Figure 6, we can offer a more general solution, which places this constraint on *every* iteration, by exploiting the loop counter. We declare a logical constant consisting of a sequence, i.e., unbounded array, of times indexed from 0. (Since this constant exists only to support timing analysis, its type does not need to be bounded in size.) Our solution also explicitly constrains the initial loop entry overheads, i.e., the '0^{th}' iteration.

Many other real-time languages are oriented to high-level schedulability analysis. In particular, the Deal language allows the time at which statements are executed to be captured, and a range of acceptable execution times for later statements to be expressed [13]. Unusually, these times denote moments *within* the execution of time-consuming statements, rather than points between statements. Also, unlike our notation, Deal timing assertions cannot refer to 'normal' program variables. However, within these limitations, programs expressed in Deal can be analysed at compile-time to construct a concurrent schedule to ensure that these constraints are met. Similarly, TOMAL [14] timing constraints are implemented by specifying task response times which are validated by the compiler. This is done by a control flow analysis of the program which detects which combinations of task states could occur during execution. The Real-Time Euclid language [15] also guarantees that its programs can be analysed for schedulability—scheduling is implemented by associating each process with a time *frame*. A similar approach is taken by High-Integrity Pearl [16].

Other systems have a philosophy of *run time* enforcement of constraints, rather than a compile-time guarantee. For example, RTC [17] is a real-time concurrency system designed on this principle. In RTC, timing constraints for behaviours are expressed as temporal scopes, and scoping is achieved by *block* statements which are inserted in C code. Real-Time Concurrent C [18] and RTC++ [19] are similar systems. The latter supports fine-grained constraints by allowing them on individual statements, but they are enforced only at run-time, and failure is handled by an exception mechanism. In FLEX [20] timing constraints are specified by *constraint blocks* which identify constraints to apply during the execution of a block of code. Failure to meet an *earliest* constraint will produce a delay, while failure to meet a *latest* constraint (i.e., a deadline) will raise an exception.

7 Future work

The concepts developed in this project are now sufficiently mature to support an implementation. Recently a preliminary study was undertaken to determine how the analysis procedures needed to support the deadline command could be incorporated into a compiler. It was shown that an annotated abstract syntax tree could potentially support the required control-flow analysis. This investigation is ongoing.

Above we described an instantiation of our notation in the SPARK language. A natural next step in this work is to develop a tool for analysing such code using our timing path analysis methodology. This would complement the SPARK Proof And Timing System [21], which also adds new annotations to SPARK, but to help with execution time prediction, rather than enforcement of real-time program properties.

So far we have assumed that a code segment containing timing constructs has dedicated access to the processor. Current work includes extension of the analysis techniques to support the deadline command in static-priority multi-tasking implementations. In this context, determining whether a deadline is met or not depends not only on the statements leading to that deadline but also on their interaction with other tasks in the system. Fortunately, real-time scheduling theory makes worst-case bounds on interference and preemption overheads readily calculable [22]. A safe (but potentially pessimistic) analysis procedure is therefore possible by factoring these figures into the timing prediction for each critical path.

8 Conclusion

We have described a small set of programming constructs for expressing fine-grained timing constraints. Most prominent is the deadline command for constraining the execution time of control paths. The constructs are flexible, formally defined, and enforceable at compile time. A practical implementation of these concepts will allow programmers of real-time applications to express hard real-time constraints, and to receive trustworthy a priori feedback on the ability of the executable code to achieve their requirements.

Acknowledgements

We wish to thank Albert Nymeyer, The University of Twente, for his work on compiler data structures to support the deadline command. The anonymous referees made several suggestions for improving this paper. This project is funded by Australian Research Council Large Grant A49937045, *Effective Real-Time Program Analysis*.

References

[1] I. J. Hayes and M. Utting. Coercing real-time refinement: A transmitter. In D. J. Duke and A. S. Evans, editors, *BCS-FACS Northern Formal Methods Workshop, 1996*, Electronic Workshops in Computing. Springer-Verlag, 1997. http://ewic.springer.co.uk/workshops/NFM96/.

[2] S. Tucker Taft and R. A. Duff, editors. *Ada 95 Reference Manual: Language and Standard Libraries*, volume 1246 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[3] S. Grundon, I. J. Hayes, and C. J. Fidge. Timing constraint analysis. In C. McDonald, editor, *Computer Science '98: Proc. 21st Australasian Computer Science Conference*, pages 575–586. Springer-Verlag, 1998.

[4] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transac-*

tions on Software Engineering, 21(7):593–604, July 1995.

[5] L. Ko, D. B. Whalley, and M. G. Harmon. Supporting user-friendly analysis of timing constraints. *ACM SIGPLAN Notices*, 30(11):99–107, November 1995.

[6] J. Barnes. *High Integrity Ada: The Spark Approach*. Addison-Wesley, 1997.

[7] A. D. Stoyenko, T. J. Marlowe, and M. F. Younis. A language for complex real-time systems. *The Computer Journal*, 38(4):319–338, 1995.

[8] A. Leung, K. V. Palem, and A. Pnueli. TimeC: A time specification language for ILP processor compilation. In K. A. Hawick and H. A. James, editors, *Proceedings of PART'98: The 5th Australasian Conference on Parallel And Real-Time Systems*, pages 57–71. Springer-Verlag, 1998.

[9] I. Lee and V. Gehlot. Language constructs for distributed real-time programming. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 57–66. IEEE, 1985.

[10] V. K. Nirkhe, S. K. Tripathi, and A. K. Agrawala. Language support for the Maruti Real-Time System. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 257–266. IEEE, December 1990.

[11] R. Gerber and S. Hong. Compiling real-time programs with timing constraint refinement and structural code motion. *IEEE Transactions on Software Engineering*, 21(5):389–404, May 1995.

[12] T. M. Chung and H. G. Dietz. Language constructs and transformation for hard real-time systems. *ACM SIGPLAN Notices*, 30(11):41–49, November 1995.

[13] D. K. Hammer, E. J. Luit, O. S. van Roosmalen, P. D. V. van der Stok, and J. P. C. Verhoosel. Dedos: A distributed real-time environment. *IEEE Parallel and Distributed Technology*, 2(4):32–47, 1994.

[14] R. B. Kieburtz and J. L. Hennessy. TOMAL—a high-level programming language for microprocessor process control applications. *ACM SIGPLAN Notices*, pages 127–134, April 1976.

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

[16] A. D. Stoyenko and W. A. Halang. Extending Pearl for industrial real-time applications. *IEEE Software*, 10(4):65–74, July 1993.

[17] V. F. Wolfe, S. Davidson, and I. Lee. RTC: Language support for real-time concurrency. *The Journal of Real-Time Systems*, 5:63–87, 1993.

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

[19] Y. Ishikawa, H. Tokuda, and C. W. Mercer. Object-oriented real-time language design: Constructs for timing constraints. Technical Report CMU-CS-90-111, Carnegie-Mellon University, March 1990.

[20] K. B. Kenny and K.-J. Lin. Building flexible real-time systems using the Flex language. *IEEE Computer*, 24(5):70–78, May 1991.

[21] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11:145–171, 1996.

[22] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.