Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Combining EDF Scheduling with occam using the Toc Programming Language

Martin KORSGAARD [1] and Sverre HENDSETH

Department of Engineering Cybernetics, Norwegian University of Science and Technology

Abstract. A special feature of the occam programming language is that its concurrency support is at the very base of the language. However, its ability to specify scheduling requirements is insufficient for use in some real-time systems. Toc is an experimental programming language that builds on occam, keeping occam's concurrency mechanisms, while fundamentally changing its concept of time. In Toc, deadlines are specified directly in code, replacing occam's priority constructs as the method for controlling scheduling. Processes are scheduled lazily, in that code is not executed without an associated deadline. The deadlines propagate through channel communications, which means that a task blocked by a channel that is not ready will transfer its deadline through the channel to the dependent task. This allows the deadlines of dependent tasks to be inferred, and also creates a scheduling effect similar to priority inheritance. A compiler and run-time system has been implemented to demonstrate these principles.

Keywords. real-time programming, occam, earliest deadline first, EDF, scheduling.

Introduction

A real-time computer system is a system which success depends not only on the computational results, but also on the time the results are delivered. The typical implementation of a real-time system is also concurrent, meaning that it has tasks that are run in parallel. While there are plenty of programming languages designed with concurrency in mind, for instance Ada [1], Java [2], occam [3] or Erlang [4], there are far fewer based on timing. Synchronous programming languages such as Esterel [5] and Lustre [6] are notable exceptions. The timing support in most other programming languages is limited to delays and access to a clock, and scheduling control is handled by specifying task priorities. A more integrated approach, where timing requirements could be specified directly by using simple language constructs, would make it easier to reason intuitively about the temporal properties of a program.

CSP [7,8] is a process algebra designed to model and reason about concurrent programs and systems. The occam programming language is largely based on CSP. The concurrency handling mechanisms of occam are one of the most fundamental parts of the language. Parallel execution is achieved using a simple PAR constructor, and is just as easy as creating serial execution (which requires the mandatory SEQ constructor). The channel data type provides native synchronous communication and synchronization between the parallel processes, the only type of inter-process communication allowed. occam has built-in language support for delays and clocks, but lacks a robust way of controlling scheduling, which can cause difficulties when implementing real-time systems [9]. One problem is that priorities are static, disabling online control over scheduling. Another problem is that certain combinations of PAR,

[1]Corresponding Author: *Martin Korsgaard, Department of Engineering Cybernetics, 7491 Trondheim, Norway*. Tel.: +47 73 59 43 76; Fax: +47 73 59 45 99; E-mail: martin.korsgaard@itk.ntnu.no.

`PRI PAR`, `ALT`, and `PRI ALT` can yield intuitively ambiguous programs. In some cases, the addition of prioritized operators can change the logical behaviour of a program [10]. An extended version of occam, called occam-π, adds many new features to occam, including process priority control [11].

Ada [1] is a programming language designed to ensure safe real-time programming for critical systems. It contains both asynchronous and synchronous concurrency functions, the latter which also is influenced by CSP. Ada allows specification of absolute task priorities. Specification of deadlines became possible from Ada 2005. The Ravenscar profile, however, which defines a safer subset of Ada more suitable to critical systems, permits neither asynchronous interruption of tasks nor synchronous concurrency. This is done to increase determinism and to decrease the size and complexity of the run-time system [12].

A recurring problem in designing a programming language that is built on timing is the lack of execution time information. In general, one cannot know in advance how long it will take to execute a given piece of code, which severely limits a system's ability to take preemptive measures to avoid missing deadlines. This also reduces the types of implementable timing requirements: An example is executing a task as late as possible before a given deadline, which is impossible without knowledge of the execution time of the task. Measurements of execution time are inadequate because execution time can vary greatly with input data. Certain features of modern computer architectures, such as caches, pipelining and branch prediction, increases the average performance of a computer, but adds complexity that makes execution times even harder to predict. Nevertheless, safe estimates of the worst-case execution time (WCET) of a program can be found using computerized tools such as aiT [13], which Airbus has used with success [14]. However, the process is inconvenient and computationally expensive, and arguably most suitable for offline schedulability analysis of safety critical systems.

Two common real-time scheduling strategies are rate-monotonic scheduling (RMS) and earliest deadline first (EDF) [15]. In RMS, task priorities are ordered by their inverse periods. If a task's relative deadline is allowed to be shorter than its period, then priorities are ordered by inverse relative deadlines instead and the algorithm is called deadline-monotonic scheduling [16]. EDF scheduling works by always executing the task with the earliest absolute deadline.

If there is not enough time to complete all tasks within their deadlines, the system is said to be overloaded. It is a common misconception that RMS is more predictable during overload that EDF, because tasks will miss deadlines in order of priority. This is not correct [17]. If a task has its execution time extended under RMS, it will affect any or all lower priority tasks in no particular order.

EDF and RMS behave differently under permanent overload. RMS will first execute tasks with a higher priority than the tasks that lead to the overload. This can result in some tasks never being executed, but ensures at least that the higher priority tasks are undisturbed. Note that priority is a function of a task's period and not its importance, so this property is not always useful. Under EDF, a task's absolute deadline will always at some point become the earliest in the system, as tasks will be scheduled even if their deadline has already passed. This means that all tasks will be serviced even if the system is overloaded. EDF has the remarkable property of doing period rescaling, where if the system has a utilisation factor $U > 1$, a task with period t_i will execute under an average period of $U \cdot t_i$ [18]. Which of the overload behaviours is the most suitable will depend on the application. For a thorough comparison of the two scheduling algorithms see [17].

In this paper we present Toc, an experimental programming language where the specification of task deadlines is at the very core of the language. In Toc, a deadline is the only reason for execution, and no code is executed without an associated deadline. Processes do not execute simply because they exist, forcing the programmer to make all assumptions on tim-

ing explicit. Scheduling is done earliest deadline first, making use of occam's synchronous channels as a way of propagating a deadline from one task to dependent tasks. Deadlines of dependent processes can thus be inferred, so that the timing requirements specified in code can be the actual requirements stemming from the specifications of the system: If a control system should do something in 10 ms, then that should appear exactly one place in the source code as "`TIME 10 MSEC`". The occam priority construct "`PRI PAR`" can then be omitted, and the result is a language that can express many types of timing requirements in an elegant manner, and where the timing requirements are clearly reflected in the source code. A prototype Toc compiler and run-time system has been implemented.

The rest of this paper is organized as follows: Section 1 describes the Toc language and gives a few examples of how to specify timing requirements using Toc. Section 2 discusses scheduling and how it is implemented. The scheduling of an example program is explained. In section 3, some of the implications of writing real-time programs in Toc are considered.

1. The Toc Language

Toc is an experimental programming language based on the idea that specification of tasks and deadlines could be fully integrated in to a procedural programming language, and furthermore, that all functionality of a program should be given an explicit deadline in order to be executed. The idea was first described in [19].

1.1. Language Specification

The language is based on occam 2.1. As with occam, statements are called processes, and are divided into primitive and constructed processes. Primitive processes are assignment, input and output on channels, `SKIP` and `STOP`. The constructive processes alter or combine primitive processes, and are made using `SEQ`, `PAR`, `IF`, `ALT`, `CASE` or `WHILE`, where the four first may be replicated using `FOR`. Definitions of the occam processes can be found in the occam reference manual [3]. Toc has the additional `TIME` constructor, which takes a relative time t, and a process P. It is defined as follows:

> The construct "TIME t : P" has a minimum allowed execution time and maximum desired execution time of t.

The maximum execution time property of `TIME` sets a deadline. It is specified only as "desired" to account for the possibility that the actual execution time exceeds t. The minimum execution time property sets a minimum time before the construct is allowed to terminate. In practice this is the earliest possible start time of processes following in sequence. This property is absolute, even in cases where it will cause another deadline to be missed. Furthermore, if a `TIME` construct follows in sequence to another, then the start time of the latter will be set to the exact termination time of the first, removing drift in ready times between consecutive tasks or task instances. The `TIME` constructor is used to create deadlines, periods and delays. Since all `ALT`s that are executed already have a timing requirement associated with them, occam's timer guards in alternations can be replaced by a `TIMEOUT` guard, which is triggered on the expiration of a deadline. Examples are shown in Table 1 and are discussed in the next section.

Central to Toc is the concept of lazy scheduling, where no code is executed unless given a deadline. With the above terminology in place, a more precise definition of the laziness of Toc can be given:

> In Toc, no primitive processes are executed unless needed to complete a process with a deadline.

Table 1. Use of the TIME constructor

#	Use	Code
1	Set deadline d milliseconds to procedure P. The TIME construct is not allowed to terminate before its deadline.	`TIME d MSEC` `  P()`
2	Delay for 2.5 seconds.	`TIME 2500 MSEC` `  SKIP`

3	Periodic process running procedure P, with deadline and period equal to 1 second.	`WHILE TRUE`
		`  TIME 1 SEC`
		`  P()`

4	Periodic process running procedure P, with deadline d and period t. Assumes d < t.	`WHILE TRUE`
		`  TIME t MSEC`
		`  TIME d MSEC`
		`  P()`

5	Sporadic process running procedure P(value) after receiving input on channel ch from another process with a deadline. The sporadic task is given deadline and minimum period d.	`WHILE TRUE`
		`  INT value:`
		`  SEQ`
		`  ch ? value`
		`  TIME d MSEC`
		`  P(value)`

6	Timeout after t microseconds waiting for input on channel ch	`TIME t USEC`
		`  ALT`
		`  ch ? var`
		`  SKIP`
		`  TIMEOUT`
		`  SKIP`

The restriction to primitive processes means that the outer layers of constructed processes are exempted from the laziness rule, and are allowed to execute until an inner primitive process. This restriction is necessary to allow the TIME constructors themselves to be evaluated. A periodic process can then be created by wrapping a TIME construct in a WHILE, without needing to set a deadline for the WHILE.

That only non-primitive processes can execute without a deadline does not imply that code without a deadline only requires insignificant execution time. For example, an arbitrarily complex expression may be used as the condition in a WHILE construct. It does mean, however, that no code with side-effects will be executed without a deadline, and consequently, that all functionality requiring input or output from a program will need a deadline.

1.2. Examples

A few examples are given in Table 1. Example one and two in Table 1 are trivial examples of a deadline and a delay, respectively. A simple periodic task with deadline equal to period can be made by enclosing a TIME block in a loop, as shown in example three. Here, the maximum time property of the TIME constructor gives the enclosed process a deadline, and the minimum time property ensures that the task is repeated with the given period. In this example the period and relative deadline are set to one second, which means that unless P misses a deadline, one instance of P will execute every second. The start time reference of each TIME construct is set to the termination time of the previous.

TIME-constructors can be nested to specify more complex tasks. The fourth example is a task where the deadline is less than the period. This requires two nested TIME constructors; the outermost ensuring the minimum period (here p), and the innermost creating the deadline

(d). The innermost maximum desired execution time takes precedence over the outermost, because it is shorter; the outermost minimum allowed execution time takes precedence over the innermost, because it is longer. In general, a timing requirement that takes n different times to specify, will take n different TIME constructors to implement.

Combining TIME with other processes, for instance an input or an IF, makes it possible to create sporadic tasks. Example five shows a sporadic task activated by input on a channel. The task runs P(value) with deadline d milliseconds after receiving value from channel ch. Example six shows a simple timeout, where the TIMEOUT guard is triggered on the expiration of the deadline.

2. Scheduling

Because Toc specifies timing requirements as deadlines, EDF was the natural choice of scheduling algorithm. A preemptive scheduler was found necessary to facilitate the response-time of short deadline tasks. A task that has the shortest relative deadline in the system when it starts will never be preempted, somewhat limiting the number of preemptions. This is a property of EDF. When the current task does not have the earliest deadline in the system, it will be preempted when the earliest deadline task becomes ready.

There are two major benefits of using EDF. The first is simply that it is optimal; EDF leads to a higher utilization than any priority-based scheduling algorithm. The second reason is the behaviour in an overload situation: With fixed-priority scheduling, a task with a long relative deadline risks never being run during overload. Because Toc has no way of specifying *importance*, assuming that shorter tasks are more important cannot be justified. EDF gives reduced service to all tasks during overload.

2.1. Discovery

A TIME constructor that can be reached without first executing any primitive processes must be evaluated immediately, because it may represent the earliest deadline in the system. The process of evaluating non-primitive processes to look for TIME constructs is called *discovery*. Discovery only needs to be run on processes where there is a chance that a TIME construct can be reached without executing any primitive processes. This limits the need for discovery to the following situations:

- At the start of a program, on procedure Main.
- After a TIME construct is completed, on processes in sequence.
- After a channel communication, on processes in sequence at both ends.
- On all branches of a PAR.

Discovery stops when a primitive process is encountered.

All TIME construct found during the same discovery will have the same ready-time, which is the time of the event that caused the discovery. If the event is the end of an earlier TIME block with time t, which deadline was not missed, then the ready time of TIME blocks found during the subsequent discovery will be precisely t later than the ready time of the first block. Thus, a periodic process such as example #3 in table 1 will have a period of exactly 1 second.

If a deadline is missed, the termination time of the TIME block and the ready time of consequent TIME blocks are moved accordingly. This also means that there is no attempt to make up for a missed deadline by executing the next instance of a task faster. This behaviour is a good choice in some real-time systems such as computer control systems or media players, but wrong where synchronization to an absolute clock is intended. In this case the task can re-synchronize itself for instance by skipping a sample and issuing a delay.

2.2. Deadline Propagation

The propagation of deadlines through channels is used to make lazy scheduling and EDF work with dependent processes. If a task with a deadline requires communication with another task, the first task will forward execution to the second to allow the communication to complete, in effect transferring its deadline to the second task. The implementation of this feature relies on a property inherited from the occam language, namely the usage rules. In occam, a variable that is written to by a process cannot be accessed by any other processes in parallel. Likewise, a channel that is used for input or output by a process cannot be used for the same by any other processes in parallel. An occam compiler must enforce these rules at compile-time. The process of enforcing the usage rules also gives the channel-ownerships, defined below:

> The input (/output)-owner of a channel is the process whose execution will lead to the next input (/output) on that channel.

The initial input- and output-owner of a channel is the first process following the declaration of the channel. The channel ownerships are updated at run-time at every PAR and end of PAR, using information gathered by the compiler during the usage rules check. With channel ownership defined, the scheduling mechanism for propagating a deadline over a channel becomes simple:

> If the current task needs to complete an input/output on a channel that is not ready, then forward execution to the output-/input-owner of that channel.

Per definition, the owner is the process whose execution will lead up to the next communication on that channel, so this forwarding is the fastest way of completing the communication. The forwarded process is executed up to the point of communication, where execution then continues from the driving side of the channel.

Usage and channel ownership is fairly simple to evaluate for scalars. With arrays, the indexing expressions may have to be evaluated at compile-time, if correct use cannot be ensured without taking the indices into account. In that case indexes will be limited to expressions of constants, literals and replicators (variables defined by a FOR). The compiler will, if necessary, simulate all necessary replicators to find the correct indices of an array used by a process.

2.3. Alternation

The alternation process ALT executes one of its actions that has a ready guard. A guard consists of a Boolean expression and/or a channel input; a guard is ready if its Boolean expression is TRUE and its channel is ready. Alternation behaves differently in Toc than in occam. In occam, the alternation process will wait if no guards are ready, but in Toc there is always a deadline driving the execution, so if there are no inputs in any guards that are ready, then the ALT must drive the execution of one of them until it, or another, becomes ready. Selecting the best alternative to drive forward is either simple or hard, depending on the deadline that drives the ALT itself.

The simple case is if the execution of the ALT is driven by a deadline from one of the guard channels, and there is no boolean guard that blocks the channel. The alternative that is the source of the deadline is then chosen. This represents programs where the ALT is a server accessed by user processes, and only the users have deadlines.

The choice is less obvious if the ALT is driven by its own deadline, or if the driving channel is not an alternative or is disabled by a boolean guard. Here, the program needs to select an alternative that allows it to proceed, ideally in some way that would aid the earliest deadline task. Unfortunately, predicting the fastest way to e.g. a complex boolean

guard may be arbitrarily difficult, thereby making it impossible to find the optimal algorithm for choosing ALT branches.

Because a perfect algorithm is not possible, it is important that the algorithm used is intuitively simple to understand for the programmer, and that the behaviour is predictable. The current implementation uses the following pragmatic decision algorithm:

1. If there are ready inputs as alternatives, choose the one with the earliest deadline. Any input that is ready has a deadline associated with it, or it would not have become ready.
2. If there is a ready guard without a channel input (just a Boolean expression that evaluates to TRUE), then choose it.
3. If there exists a channel input alternative that is not disabled by a boolean guard, forward execution to the output-owner of the channel, but do not select the alternative. This is because execution could require input on another alternative of the same ALT, which would cause a deadlock if the first alternative was already selected. At some point, execution will be forwarded back to the ALT, now with an input ready. That input is then selected.
4. If no alternatives are unguarded, act as a STOP. A STOP in Toc effectively hangs the system, because it never terminates but retains the earliest deadline.

2.4. Deadline Inversion

In general, when synchronizing tasks, a task may be in a blocked state where it is not allowed to continue execution until another task has completed some work. The naïve way to schedule such a system is to ignore any blocked tasks and schedule the rest of the tasks normally. This leads to a timing problem known as unbounded priority inversion.

Say the highest priority task is blocked waiting for the lowest priority task. This is a simple priority inversion, and cannot be avoided as long as high and low-priority tasks share resources. The unbounded priority inversion follows when the scheduler selects the second highest priority task to run, leaving the lowest priority task waiting. Now the highest priority task will remain blocked. In effect, all tasks now take precedence over the one with the highest priority.

A reformulation of the problem is that the scheduler does not actively help to execute its most urgent task. One way to alleviate the problem is to use priority inheritance [20]. Using priority inheritance, if a high priority task is blocked waiting for a lower priority task; the lower priority task will inherit the priority of the blocked task, thus limiting the priority inversion to one level. In a sense, the lower priority task completes its execution on behalf of the higher priority task. Priority inheritance has a number of weaknesses; in particular it does not work well with nested critical regions [21]. Other schemes exist, for instance the priority ceiling protocol [22] and the stack resource policy [23].

The Toc notion of priority inversion — or deadline inversion — is different than the one used in classical scheduling. Classical priority inversion is defined for lockable resources with well-defined owners, where the locking process is always the process that will open it later. This property does not apply to systems synchronized by channels. Also, in Toc, tasks are meant to drive the execution of their dependencies. In principle, when communication is done on channels there is a simple deadline inversion every time the earliest deadline task needs to communicate and the other side is not ready. However, this is an inversion by design, and not an unfortunate event.

A situation more similar to a classical priority inversion is when two tasks communicate with a third server, which could represent a lockable resource. If the earliest deadline task is not ready, then the second task may set the server in a state where it is unable to respond to the earliest deadline task when it becomes ready. A third task, with a deadline between that

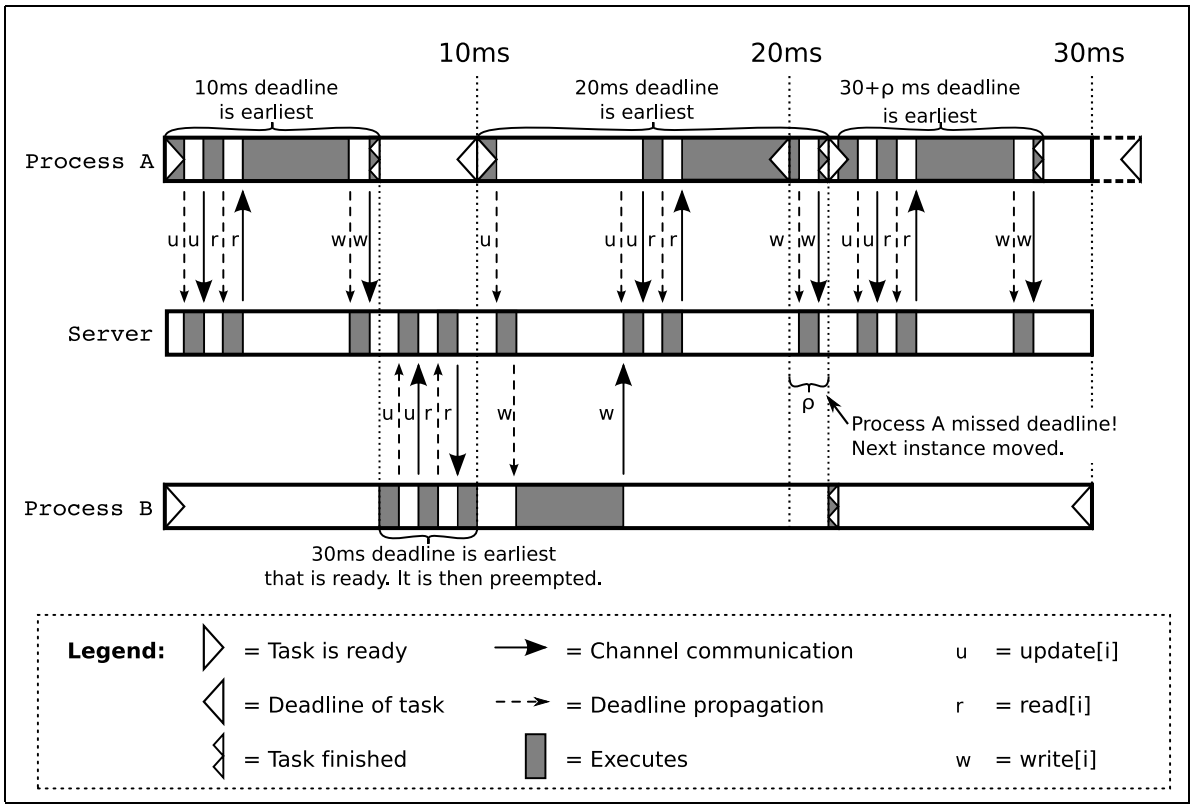

Figure 1. Timeline of scheduling example. Approximate time scale.

of the first and second task will then indirectly block a task with a shorter deadline, possibly leading to unbounded priority inversion.

In Toc, the deadline propagation rule automatically resolves these situations. The earliest deadline process will never be blocked, rather it will transfer its execution and deadline to the blocking processes, so that they execute as if with an earlier deadline. This effect is similar to priority inheritance: With priority inheritance, the blocking process inherits the priority of the blocked process; with deadline propagation, the blocked process will transfer its deadline to the blocking process.

2.5. Scheduling Example

An example of the scheduling of a system with a server and two users is shown in Figures 1 and 2. A server allows a variable to be updated concurrently. Two periodic user tasks access the server. If the server is not ready when the earliest deadline task needs it, then it will finish its current transaction driven by the earliest deadline, the same way it would have executed with a higher priority if priority inheritance was used. A brief explanation is given below:

1. The program starts and executes up to the first primitive processes in all three parallels. Two parallel deadlines are discovered. The user process with the short deadline will be referred to as process A, the other as process B.
2. Process A starts because it has the earliest deadline. It eventually needs to output on channel `update[0]`. The input-owner of the channel-array is the `Server` process, so execution is forwarded to the server process through deadline propagation.
3. The server executes up to the communication and zero is sent over the channel.
4. Process A now needs to input from `read[0]`. First it executes up to the communication and then execution is forwarded to the server. The server executes up to the communication and `value` is sent over the channel.

```
PROC Server(CHAN[2] INT update?, read!, write?)
  WHILE TRUE
    INT dummy, value:
    ALT i = 0 FOR 2
      update[i] ? dummy
        SEQ
          read[i] ! value
          write[i] ? value

PROC User(VAL INT period, CHAN INT update!, read?, write!)
  WHILE TRUE
    TIME period MSEC
      INT x:
      SEQ
        update ! 0
        read ? x
        WORK 6 MSEC      -- Pseudo-statement for requiring 6 ms of CPU time.
        write ! x+id
        WORK 500 USEC    -- Do some clean-up to finish the task

PROC Main()
  CHAN[2] INT update, read, write:
  PAR
    User(10, update[0], read[0], write[0])  -- Process A
    User(30, update[1], read[1], write[1])  -- Process B
    Server(update, read, write)
```

Figure 2. Code for scheduling example

5. Process A works. Some time later it writes the new value to the server through the `write` channel, and finishes its task.
6. Process B begins in the same way. However, at $t = 10$ms it is preempted by process A, whose task is now ready and has the earliest deadline.
7. Now process B has blocked the server. This is a priority inversion in the classical sense. Process A proceeds as last time, forwarding execution to the input-owner of `update[0]`, which is still the server. To proceed, the server must output on `write[1]`, which is not ready, and forwards execution to the input-owner of that channel (process B).
8. Process B executes up to the output on `write[1]`, driven by the server's need to input on `write[1]`, which again is driven by process A's need to output on `update[0]`. This frees the server and allows process A to continue.
9. Notice that the second instance of process A's task misses its deadline. It has its new period offset accordingly. Also notice that only then is process B allowed to finish.

3. Implications

In occam, a `PRI PAR` or `PRI ALT` can be used to affect the scheduling of parallel processes. This can potentially affect the logical behaviour of a program when modelling the program with CSP, as certain program traces are no longer possible. A `TIME` constructor will also restrict possible traces of a Toc program by always preferring execution of the earliest deadline task to others. The effect of `TIME` constructors on the logical behaviour of a Toc program is much greater than the effect of prioritized constructs in occam programs. For example removing all `TIME` constructors will make any Toc program equal to `STOP`.

Adding a process may also make a program behave as `STOP`. Take the example given in Figure 3. This may be an attempt to initialize a variable before starting a periodic process, but the primitive assignment makes the entire program behave as `STOP`. There is no deadline for

```
PROC Main()
  INT a:
  SEQ
    a := 42
    WHILE TRUE
      TIME 100 MSEC
        P(a)
```

Figure 3. Dead code example. Lack of timing requirements on the assignment makes the program equal to `STOP`.

```
task body Periodic_Task is
  Period       : Time_Span := Milliseconds(30);
  Rel_Deadline : Time_Span := Milliseconds(20);
  Next         : Ada.Real_Time.Time;
begin
  Next := Ada.Real_Time.Clock;
  Set_Deadline(Next+Rel_Deadline);
  loop
    delay until Get_Deadline;
    Action;
    Next := Next + Interval;
    Set_Deadline(Next+Rel_Deadline);
    delay until Next;
  end loop;
end Periodic_Task;
```

```
WHILE TRUE
  TIME 30 MSEC
    TIME 20 MSEC
      Action()
```

Figure 4. Comparison of a periodic process in Ada and Toc. Left: Ada. Right: Toc. Ada example is a modified example from [25]

.

executing the assignment and therefore it will never happen. The `TIME` constructor that follows will never be discovered and the periodic process will not start. In Toc, the programmer must consider the deadlines of all functionality in the system, no matter how trivial. Because assignments, like the one in Figure 3, are typically quite fast, not adding a timing requirement signals that the programmer does not care exactly how long time the assignment will take. In Toc, not caring is not an option, and deadlines are always required. The compiler will in many cases issue a warning to avoid such mistakes.

It has been argued that it is more awkward to assign arbitrary deadlines to tasks than to assign arbitrary priorities [24]. This is debatable, but it may anyway be easier to find a *non*-arbitrary deadline than a priority: Many seemingly background tasks can be given sensible deadlines: A key press does not need to give visual feedback faster than it is possible for the human eye to perceive it. A control monitor in a process plant will need some minimum update frequency in order to convey valid information to the operators. Setting a deadline for the latter task, however arbitrary, is a step up from giving it a low priority under fixed-priority scheduling, where a scheduling overload could effectively disable the task.

4. Conclusions and Future Work

This paper presented the language Toc, which allows the specification of deadlines in the source code of a program. This is done using the `TIME` constructor, which provides elegant language support for specifying deadlines and tasks. A periodic task with a deadline can be implemented with just a few lines of code, compared to the rather more complex construct required in for example Ada, as shown in Figure 4. The combination of EDF and deadline propagation yields a simple and versatile scheduling strategy. Lazy scheduling forces the programmer to consider all timing requirements in the system, not only those that are considered

real-time in the classical sense. This may potentially increase awareness of timing requirements for parts of real-time systems for which such requirements were previously ignored, such as sporadic or background tasks and error handling.

The prototype compiler was developed in Haskell, using the parser generator `bnfc` [26]. The compiler generates C code, which can then be compiled with an ordinary C compiler. The current run-time system is written in C, and includes a custom scheduler using POSIX threads. The scheduler is linked into the program, and the resulting executable can be run as an application under another operating system. Both the compiler and run-time system are prototypes under development, and have so far only been used to test small programs, though quite successfully so.

The next task would be to test Toc on a real-time system of some complexity, to see if the `TIME` constructor presented here is practical and suitable for such a task. Of particular interest is seeing how many timing requirements that are actually necessary in the specification of such a system, when no execution is allowed without one.

References

[1] J. D. Ichbiah, B. Krieg-Brueckner, B. A. Wichmann, J. G. P. Barnes, O. Roubine, and J.-C. Heliard, "Rationale for the design of the Ada programming language," *SIGPLAN Not.*, vol. 14, no. 6b, pp. 1–261, 1979.

[2] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 2000.

[3] SGS-THOMPSON Microelectronics Limited, *occam® 2.1 Reference Manual*, 1995.

[4] J. Armstrong and R. Virding, "ERLANG – an experimental telephony programming language," *Switching Symposium, 1990. XIII International*, vol. 3, 1990.

[5] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "LUSTRE: A declarative language for programming synchronous systems," *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, 1987.

[7] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, pp. 666–677, 1978.

[8] S. Schneider, *Concurrent and Real Time Systems: The CSP Approach*. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[9] D. Q. Z. C. Cecati and E. Chiricozzi, "Some practical issues of the transputer based real-time systems," *Industrial Electronics, Control, Instrumentation, and Automation, 1992. Power Electronics and Motion Control., Proceedings of the 1992 International Conference on*, pp. 1403–1407 vol.3, 9-13 Nov 1992.

[10] C. J. Fidge, "A formal definition of priority in CSP," *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 4, pp. 681–705, 1993.

[11] P. Welch and F. Barnes, "Communicating mobile processes: introducing occam-pi," in *25 Years of CSP* (A. Abdallah, C. Jones, and J. Sanders, eds.), vol. 3525 of *Lecture Notes in Computer Science*, pp. 175–210, Springer Verlag, Apr. 2005.

[12] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the Ada Ravenscar Profile in high integrity systems," *ACM SIGAda Ada Letters*, vol. 24, no. 2, pp. 1–74, 2004.

[13] R. Heckmann and C. Ferdinand, "Worst case execution time prediction by static program analysis," *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pp. 125–, 26-30 April 2004.

[14] J. Souyris, E. L. Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann, "Computing the worst-case execution time of an avionics program by abstract interpretation," in *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) analysis*, pp. 21–24, 2005.

[15] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[16] J. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

[17] G. C. Buttazzo, "Rate monotonic vs. EDF: Judgment day," *Real-Time Syst.*, vol. 29, no. 1, pp. 5–26, 2005.

[18] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén, "Feedback-feedforward scheduling of control tasks," *Real-Time Systems*, no. 23, pp. 25–53, 2002.

[19] M. Korsgaard, "Introducing time driven programming using CSP/occam and WCET estimates," Master's thesis, Norwegian University of Science and Technology, 2007.
[20] D. Cornhilll, L. Sha, and J. P. Lehoczky, "Limitations of Ada for real-time scheduling," *Ada Lett.*, vol. VII, no. 6, pp. 33–39, 1987.
[21] V. Yodaiken, "Against priority inheritance," tech. rep., FSMLabs, 2002.
[22] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.
[23] T. Baker, "A stack-based resource allocation policy for realtime processes," *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pp. 191–200, Dec 1990.
[24] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*. Essex, England: Pearson Education Limited, third ed., 2001.
[25] A. Burns and A. J. Wellings, "Programming execution-time servers in Ada 2005," *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pp. 47–56, Dec 2006.
[26] M. Pellauer, M. Forsberg, and A. Ranta, "BNF converter: Multilingual front-end generation from labelled BNF grammars," tech. rep., 2004.