Design Patterns for Communicating Systems with Deadline Propagation

Martin KORSGAARD [a,1] and Sverre HENDSETH [a]
[a] *Department of Engineering Cybernetics, Norwegian University of Science and Technology*

Abstract. Toc is an experimental programming language based on occam that combines CSP-based concurrency with integrated specification of timing requirements. In contrast to occam with strict round-robin scheduling, the Toc scheduler is lazy and does not run a process unless there is a deadline associated with its execution. Channels propagate deadlines to dependent tasks. These differences from occam necessitate a different approach to programming, where a new concern is to avoid dependencies and conflicts between timing requirements. This paper introduces client-server design patterns for Toc that allow the programmer precise control of timing. It is shown that if these patterns are used, the deadline propagation graph can be used to provide sufficient conditions for schedulability. An alternative definition of deadlock in deadline-driven systems is given, and it is demonstrated how the use of the suggested design patterns allow the absence of deadlock to be proven in Toc programs. The introduction of extended rendezvous into Toc is shown to be essential to these patterns.

Keywords. real-time programming, Toc programming language, design patterns, deadlock analysis, scheduling analysis

Introduction

The success of a real-time system depends not only on its computational results, but also on the time at which those results are produced. Traditionally, real-time programs have been written in C, Java or Ada, where parallelism and timeliness relies on the concepts of tasks and priorities. Each timing requirement from the functional specification of the system is translated into a periodic task, and the priority of each task is set based on a combination of the tasks period and importance. Tasks are implemented as separate threads running infinite loops, with a suitable delay statement at the end to enforce the task period. Shared-memory based communication is the most common choice for communication between threads. With this approach, the temporal behaviour of the program is controlled indirectly, through the choice of priorities and synchronization mechanisms.

The occam programming language [1] has a concurrency model that only supports synchronous communication, and where parallelism is achieved using structured parallels instead of threads. Control of timeliness is only possible with a prioritized parallel construct, which is insufficient for many real-time applications [2]. The occam-π language [3] improves on this by allowing control of absolute priorities. The occam process model is closely related to the CSP process algebra [4, 5], and CSP is well suited for formal analysis of occam programs.

Toc [6, 7] is an experimental programming language based on occam that allows the specification of timing requirements directly in the source code. The goal of Toc is to create a

[1]Corresponding Author: *Martin Korsgaard, Department of Engineering Cybernetics, O.S Bragstads plass 2D, 7491 Trondheim, Norway*.
Tel.: +47 73 59 43 76; Fax: +47 73 59 45 99; E-mail: martin.korsgaard@itk.ntnu.no.

language where the temporal response is specified explicitly rather than indirectly, and where the programmer is forced to consider timing as a primary concern, rather than something to be solved when the computational part of the program is complete. A prototype compiler and run-time system has been developed.

In Toc, the timing requirements from the specification of the system can be expressed directly in source code. This is achieved with a dedicated deadline construct and synchronous channels that also propagate deadlines to dependent tasks. A consequence of deadline propagation is an implicit deadline inheritance scheme.

The Toc scheduler is *lazy* in that it does not execute statements without an associated deadline. Server processes or other shared resources do not execute on their own, but must be driven by deadlines implicitly given by their access channels. The laziness means that temporal requirements cannot be ignored when programming, and tasks that traditionally would be implemented as deadline-less "background tasks" must now be given explicit deadlines. occam programs use a strict, round-robin scheduler that closely matches CSP's principle of maximum progress. This is one reason why CSP is well suited to model occam programs. However, laziness is not closely matched by a principle of maximum progress, and that makes CSP-based analysis of Toc programs harder.

For all parallel systems that use synchronization there is a risk of deadlock. In occam, the *client-server* design paradigm can be used to avoid deadlocks by removing the possibility of a circular wait [8, 9]. Here, each process acts as either a client or a server in each communication, and each communication is initiated by a request and terminated by a reply. If the following three criteria are met, then the system is deadlock-free:

1. Between a request to a server and the corresponding reply, a client may not communicate with any other process.
2. Between accepting a request from a client and the corresponding reply, a server may not accept requests from other clients, but may act as a client to other servers.
3. The client-server relation graph must be acyclic.

An alternative deadlock-free design pattern described in [8] is I/O-PAR, guaranteed when all processes proceed in a sequential cycle of computation followed by all I/O in parallel. The whole system must progress in this fashion, similar to *bulk synchronous parallelism* (BSP) [10]. It is also possible to use a model checker such as SPIN [11] or FDR2 [12] to prove the absence of deadlocks under more general conditions.

For some real-time systems, proving the absence of deadlocks is not sufficient, and a formal verification of schedulability is also required. Toc is scheduled using earliest deadline first (EDF), and for EDF systems, a set of *independent*, periodic tasks with deadline equal to period is schedulable if and only if:

$$\sum_i \frac{C_i}{T_i} \leq 1 \tag{1}$$

where C_i is the worst-case computation time (WCET) of task i, and T_i is the period of that task [13]. Equation 1 ignores overheads associated with scheduling.

Scheduling analysis is more complicated for task sets that are not independent. One effect that complicates analysis of dependent tasks is *priority inversion* [14], where a low priority task holding a resource will block a high priority task while holding that resource. Priority inversions are unavoidable in systems where tasks share resources. In contrast, an *unbounded priority inversion* is a serious error, and occurs when the low priority task holding the resource required by the high priority task is further preempted by intermediate priority tasks, in which case the high priority task may be delayed indefinitely. Solutions to this problem include the priority inheritance and the priority ceiling algorithms, both defined in [15]. These algorithms are restricted to priority-based systems. Other systems can be scheduled

and analyzed by using the Stack Resource Policy [16], for example. SRP supports a wide range of communication and synchronization primitives, but is otherwise similar to the priority ceiling algorithm. Unfortunately, schedulability analysis of task sets that communicate synchronously using rendezvouses are not well developed [17].

If deadlines may be shorter than periods, necessary and sufficient schedulability conditions can be found using the Processor Demand Criterion [18], but the computation is only practical if the relative start-times of all the tasks are known (a "complete task set"). If the relative start-times are not known (an "incomplete task set") then the problem of determining schedulability is NP-hard.

To use these analysis methods one needs to know the WCET C of each task. This is far from trivial, and may be impossible if the code contains variable-length loops or recursions. Fortunately, programmers of real-time systems usually write code that can easily be found to halt, reducing the problem to finding out when. Execution times vary with input data and the state of the system, meaning that measurements of the execution time are not sufficient. The use of modern CPU features such as superscalar pipelines and cache further increases this variation.

Despite of these difficulties, bounds on execution times can still be found in specific cases using computerized tools such as aiT [19]. The process is computationally expensive [20] and subject to certain extra requirements such as manual annotations of the maximum iterations in a loop [21]. AiT is also limited to fixed task sets of fixed priority tasks.

This paper discusses how analysis of schedulability and deadlocks can be enabled in Toc by using a small set of design patterns. The paper begins with a description of the Toc timing mechanisms in Section 1 and the consequences of these mechanisms. Section 2 describes deadlocks in deadline propagating systems, and how it differs from deadlocks in systems based on the principle of maximum progress. Section 3 describes the suggested design patterns and how they are used to program a system in an intuitive manner that helps avoid conflicts between timing requirements. Section 4 explains how schedulability analysis can be done on systems that are designed using these design patterns. The paper ends with a discussion and some concluding remarks.

1. Temporal Semantics of Toc

The basic timing operator in Toc is `TIME`, which sets a deadline for a given block of code, with the additional property that the `TIME` construct is not allowed to terminate before the deadline. The deadline is a scheduling guideline, and there is nothing in the language to stop deadlines from being missed in an overloaded system. In contrast, the minimum execution time constraint is absolute and is strictly enforced.

The definition of a *task* in Toc is the process corresponding to a `TIME` construct. A periodic task with deadline equal to period is simply a `TIME` construct wrapped in a loop, and a periodic task with a shorter relative deadline than period is written by using two nested `TIME` constructors: the outer restricting the period while the inner sets the deadline. Toc handles the timing of consecutive task instances in a way that removes drift [6, 7]. Sporadic tasks can be created as a task triggered by a channel by putting the `TIME` construct in sequence after the channel communication.

A `TIME` construct will drive the execution of all statements in its block with the given deadline whenever this deadline is the earliest in the system. If channel communications are part of that block, then the task will also drive the execution of the other end of those channels up to the point where the communications can proceed. A task has *stalled* if it cannot drive a channel because the target process is waiting for a minimum execution time to elapse, but execution of the task will continue when the target becomes ready.

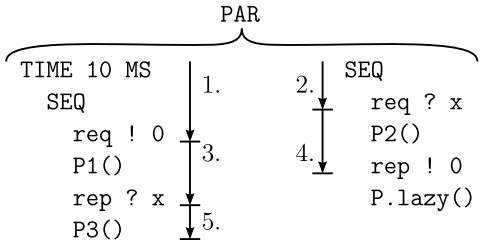

Figure 1. Order of Execution when task communicates with passive process.

```
WHILE TRUE                    WHILE TRUE
  SEQ                           input ?? x
    input ? x                     output ! x
    output ! x
```

(a) (b)

Figure 2. Use of Extended Rendezvous. (a) `x` will not be passed on due to laziness. (b) `x` will be forwarded with the deadline inherited from `input`.

The definition of laziness in Toc is that primitive processes (assignment, input/output, `SKIP` and `STOP`) not driven by a deadline will not be executed, even if the system is otherwise idle. However, constructed processes will be evaluated even without a deadline in cases where a `TIME` construct can be reached without first executing primitive processes.

Data from the compiler's parallel usage rules checker is used to determine which process to execute to make the other end of a given channel ready. An example of the order of execution is given in Figure 1. Here, the process `P.lazy()` will not be executed because it is never given a deadline.

1.1. Timed Alternation

In Toc, alternations are not resolved arbitrarily, but the choice of alternative is resolved by analyzing the timing requirements of the system.

Choice resolution is straight forward when there is no deadline driving the `ALT` itself, so that the only reason that the process with the `ALT` is executing at all is that it is driven by one of its guard channels, which by the definition of EDF must have the earliest deadline of all ready tasks in the system at that time. By always selecting the driving alternative it is ensured that the choice resolution always benefits the earliest deadline task in the system. This usage corresponds to the `ALT` being a passive server with no timing requirements on its own, that handles requests from clients using the deadlines of the most urgent client.

Other cases cannot be as easily resolved: If the `ALT` itself has a deadline, if the `ALT` is driven by a channel not part of the `ALT` itself or if the driving channel is blocked by a Boolean guard, then the optimal choice resolution with respect to timing cannot easily be found. If the design patterns presented in this paper are used, then none of these situations will arise, and therefore these issues are not discussed here.

1.2. Use of Extended Rendezvous

A consequence of laziness is that driving a single channel communication with another process cannot force execution of the other process beyond what is required to complete the communication. One consequence of this is that it is difficult to write multiplexers or simple data forwarding processes.

This is illustrated in Figure 2a: if something arrives on `input` then it *will not* immediately be passed on to `output` because there no deadline driving the execution of the second communication. The output will only happen when driven by the deadline of the next input.

A workaround is to add a deadline to the second communication, but this is awkward, and the choice of deadline will either be arbitrary or a repeated specification of an existing deadline, neither of which is desirable. Another solution, though ineffective, is to avoid forwarding data altogether, but this would put severe restrictions on program organization.

Extended inputs [3] fix the problem. An extended input is a language extension from occam-π that allows the programmer to force the execution of a process "in the middle" of a channel communication, after the processes have rendezvoused but before they are released. Syntactically, this *during process* is indented below the input, and a double question-mark is used as the input operator. In Toc, the consequence is that the deadline used to drive the communication will also drive the during process.

A correct way to forward a signal or data is shown in Figure 2b.

2. Deadlock in Timed Systems

A deadlock in the classical sense is a state where a system cannot have any progress because all the processes in the system are waiting for one of the other processes. For a system to deadlock in this way there has to be at least one chain of circular waits.

In Toc, when the earliest-deadline task requires communication over a channel it will pass its deadline over to the process that will communicate next on the other side of that channel. If this process in turn requires communication with another process, the deadline will propagate to this other process as well. Thus a task that requires communication over a channel is never blocked; it simply defers its execution to the task that it is waiting for. This can only fail if a process — through others — passes a deadline onto itself. In that case the process cannot communicate on one channel before it has communicated on another, resulting in a deadlock. This leads to the following alternative definition of deadlock in Toc:

Definition 1 (Deadlock) *A system has deadlocked if and only if there exists a process that has propagated its deadline onto itself.*

One advantage of this definition is that it takes laziness into account. Deadlock avoidance methods that ignore laziness are theoretically more conservative, because it is possible to construct programs that would deadlock but does not because the unsafe code is lazy and does not execute.

In occam one can avoid deadlocks by the client-server paradigm if clients and servers follow certain rules and the client-server relation graph is acyclic. This ensures that there are no circular waits. In Toc, one can use the same paradigm to avoid deadlocks by ensuring that there is no circular propagation of deadlines.

In Toc there can also be another type of deadlock-like state, the *slumber*, where a system or subsystem has no progress because it has no deadlines. Slumber occurs e.g. in systems that are based on interacting sporadic tasks, where sporadic tasks are triggered by other tasks. If a trigger is not sent, perhaps because of an incorrectly written conditional, the system may halt as all processes wait for some other process to trigger them. Even though the result is similar to a deadlock — a system doing nothing — the cause of a slumber is laziness and incorrect propagation of deadlines rather than a circular wait.

3. Design Patterns

By essentially restricting the way a language is used, design patterns can be used to make a program more readable and to avoid common errors [22]. The goals of the design patterns for Toc presented in this paper are:

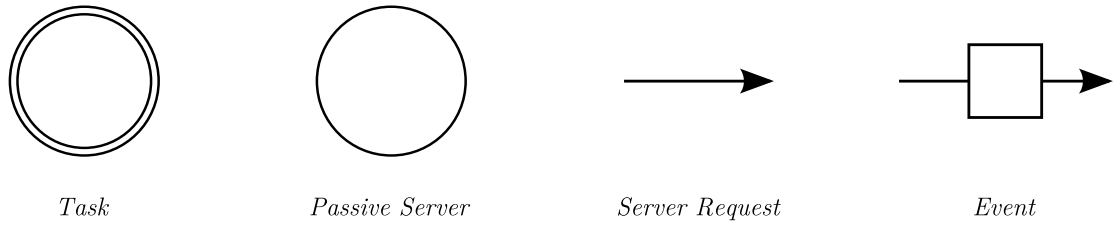

Figure 3. Diagram symbols for design patterns. An arrow points in the direction of deadline propagation, not data flow.

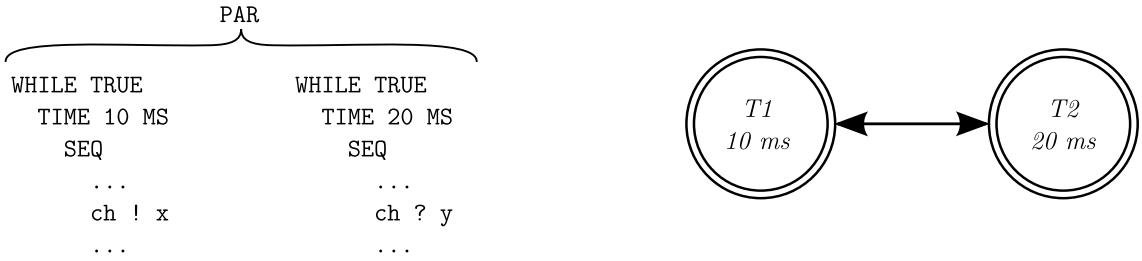

Figure 4. A task driving a task distorts timing.

1. To be able to reason on the absence of deadlocks in a system.
2. To allow for schedulability analysis.
3. To provide a flexible way of implementing the desired real-time behaviour of a system.

The design patterns should be designed in such a way that knowledge of a program at the pattern level should be sufficient to do both deadlock and schedulability analysis, so that analysis does not need to take further details of the source code into account.

The design patterns will be compatible with the client-server paradigm, so that this can be used to avoid deadlocks. Schedulability analysis is done by analyzing the timing response of each pattern separately, and then using the deadline propagation graph of the system at the pattern level to combine these timing responses into sufficient conditions for schedulability.

3.1. The Task

A task is any process with a `TIME` construct, and is depicted with a double circle as shown in Figure 3. All functionality of the system must be within the responsibility of a task.

Tasks are not allowed to communicate directly with one another, because this can lead to distorted timing. This is illustrated by the two tasks in Figure 4: *T1* has a deadline and period of 10 ms and *T2* of 20 ms, which means that *T1* wants to communicate over the channel once in every 10 ms period, while *T2* can only communicate half as often. A task's minimum period is absolute, so *T1*, which was intended to be a 10 ms period task, will often have a period of 20 ms, missing its deadline while stalled waiting for *T2*. Moreover, *T2* will often execute because it is driven by *T1* and not because of its own deadline, in which case it is given a higher scheduling priority than the specification indicates. The result of direct communication in this case is that two simple, independently specified timing requirements become both dependent and distorted.

3.2. The Passive Server

A process is passive if it has no deadline of its own. Thus a *passive server* is driven exclusively by deadlines inherited from clients through its access channels. A passive server is depicted as a single circle, and communications between a task and a server is shown as an arrow

from the driving client to the server. This is usually, but not necessarily, the same as the data direction.

The protocol for an access to a server may involve one or more communications. The first communication is called the request, and any later communications are called replies. To follow the client-server paradigm, if the protocol involves more than one communication, the client cannot communicate between the request and the last reply, and the server may not accept requests from other clients in this period. Again, the server is allowed to perform communications with other servers as a client, propagating the deadline further on.

The possibility of preemption means that the temporal behaviour of tasks using the same server are not entirely decoupled: If a task that uses a server preempts another task holding the same server, then the server may not be in a state where it can immediately handle the new request. In this case, the deadline of the new request must also drive the handling of the old request, adding to the execution requirements of the earlier deadline task. This is equivalent to priority inversion in preemptive, priority-based systems. The preempted task executes its critical section with the earlier task's deadline, resolving the situation in a manner analogous to priority inheritance.

3.3. Sporadic Tasks and Events

A passive server is useful for sharing data between tasks, but because a server executes with the client's deadline, a different approach is needed to start a periodic tasks that is to execute with its own deadline. A communication to start a new task will be called a *trigger*; the process transmitting the trigger is called the *source* and the sporadic task being triggered is called the *target*.

A simple way to program a sporadic task would be to have a `TIME` construct follow in sequence after a channel communication. When another task drives communication on the channel, the `TIME` construct will be discovered and the sporadic task will be started. However, this approach is problematic, for reasons similar to why tasks should not communicate directly. A target sporadic task that has been started will not accept another trigger before its deadline has elapsed; i.e. if the target task has a relative deadline D then the source must wait at least D between each trigger to avoid being stalled. If a new trigger arrives too early, the source will drive the target and eventually stall.

This illustrates the need for *events* that can be used to start a sporadic task, without the side-effects of stalling the source or driving the target. Events must appear to be synchronous when the target is ready and asynchronous when the target is not: If the target is waiting for a trigger it must be triggered synchronously by an incoming event, but if it the target is busy then incoming events must not require synchronization.

The *event pattern* is a way of implementing this special form of communication in Toc. An event pattern consists of an *event process* between the source and the target, where the communication between the event process and the target must follow certain rules. An event process is depicted in diagrams as an arrow through a square box.

An event process forwards a trigger from the source to the target when the target is ready, and stops it from being forwarded when the target is busy. The event process must therefore know the state of the target. After a trigger is sent to the target then the target can be assumed to be busy, but the target must explicitly communicate to the event process when it becomes ready. The event and target processes must be together follow two rules:

1. After the target notifies the event process that is ready then the target cannot communicate with other processes before triggered by a new event.
2. A trigger from a source should never drive the target to become ready. This implies that sending a trigger will never stall the source.

Because of these rules, a deadline from the source will never propagate beyond the target, so that the event process is never involved in a circular propagation of a deadline. This in turn means that an event can never be the source of a deadlock in the super-system, even if communications with the event process makes the super-system cyclic. It is therefore safe to disregard events when using the client-server paradigm to avoid deadlocks, or equivalently, to model an event as a passive server to both the source and the target. This sets an event apart from other processes, which is emphasized by the use of a square diagram symbol instead of a circle.

There are many possible designs for an event process: it may discard the input when the target is not ready or it may buffer the input. It may have multiple inputs or outputs, and it may also hold an internal state to determine which events to output.

4. Schedulability Analysis

Traditionally, threads execute concurrently and lock resources when required, so that other threads are blocked when trying to access those resources. On the other hand, in Toc, the earliest deadline task is always driving execution, but the code being executed may belong to other tasks. A task is never blocked. This means that the schedulability analysis can assume that tasks are independent, but that the WCET for each task will then have to include the execution time of all the parts of dependent processes it may also have execute.

All sporadic tasks will here be assumed to execute continuously with their minimum periods. This is a conservative assumption that enables schedulability analysis on a pure pattern level, without having to analyze actual source code to find out which sporadic tasks that may be active at the same time. To find the worst-case execution time, each task will also be assumed to be preempted by all other tasks as many times as is possible.

The analysis will also be simplified by assuming that $D = T$ for all tasks, which in code terms means that there are no nested `TIME` constructs. Because no assumption can be made as to the relative start-time of tasks in Toc, this restriction is necessary to make the schedulability analysis problem solvable in polynomial time. In this setting, the schedulability criterion is simply the standard EDF criterion given in Equation 1. The relative deadlines $D_i = T_i$ are read directly from code, so the main problem of the schedulability analysis is to find C_i: the worst-case execution time for each task including execution of other tasks due to deadline propagation. This variable is made up of three components:

1. The base WCET if not preempting and not being preempted.
2. The WCET penalty of being preempted by others.
3. The WCET penalty of preempting other processes.

Dependencies between processes will be analyzed using the deadline-propagation graph of the system. An example graph in given in Figure 5.

4.1. WCET of a Server or Event

The execution time of a server is complex due to preemption. If a server protocol involves at least one reply, then a client waits for the server in one of two phases: Waiting for the server to accept its request, and waiting for the server up to the last reply. In the first phase the server is ready to accept any client, and may switch clients if preempted, but in the second phase it will complete the communication with the active client even if preempted (executing with the deadline of the preempting task). If the server handles the request by an extended rendezvous then this is equivalent to a second phase even if the server has no reply, as the server cannot switch clients during an extended rendezvous.

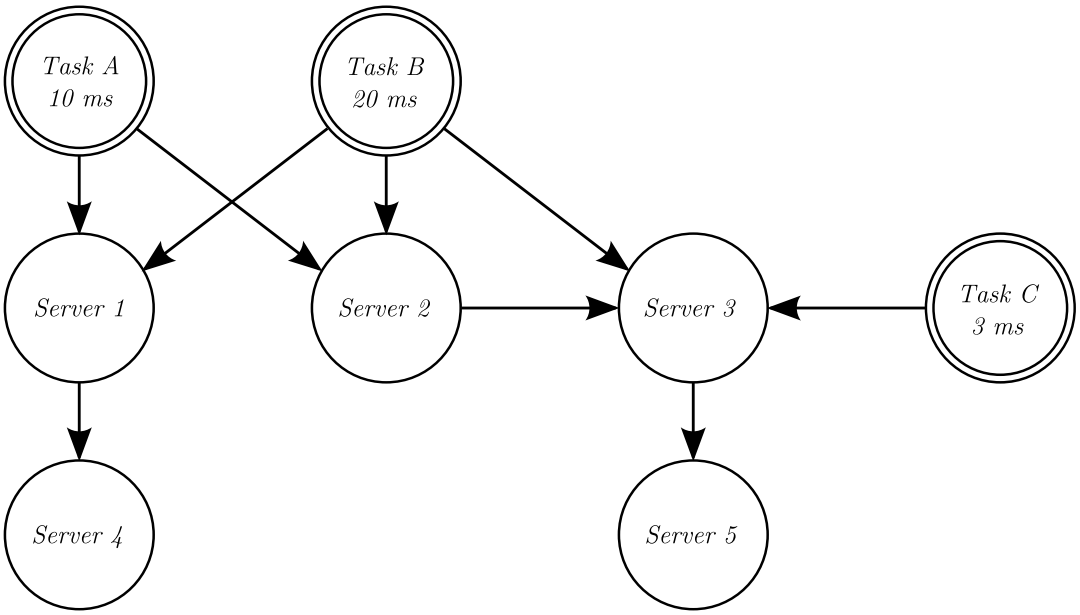

Figure 5. Example deadline propagation graph with tasks and servers.

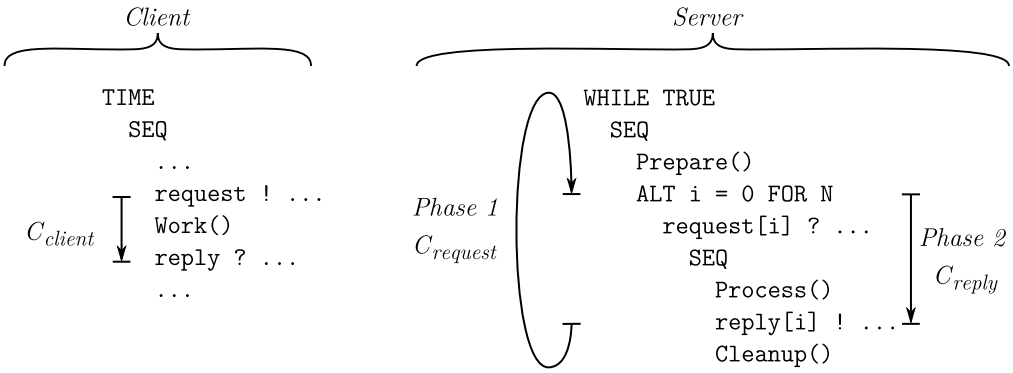

Figure 6. Timing of passive server

The WCETs for the two phases of a protocol with one reply are illustrated in Figure 6. For a server s, the WCET of the server in the first phase is denoted $C_{s,request}$ and consists of lazy post-processing of the previous access or initialization of the server. The WCET of the server in the second phase is denoted $C_{s,reply}$, which is zero if there is no second phase. $C_{s,reply}$ consists in part of the WCET of code local to the process of the server, $\widehat{C}_{s,reply}$, but also of execution used by other servers accessed by s. If $\operatorname{acc} s$ is the set of servers accessed directly by s, then

$$C_{s,reply} = \widehat{C}_{s,reply} + \sum_{s' \in \operatorname{acc} s} (C_{s',request} + C_{s',reply}) \tag{2}$$

This is a recursive formula over the set of servers. As long as the deadline-propagation network is acyclic, then the recursion will always terminate as there will always be a server at the bottom where $\operatorname{acc} s = \emptyset$. It is assumed that each server accesses another server directly only once for each of its own requests, but the calculations can easily be adjusted to accommodate multiple accesses by simply adding the WCET of those servers multiple times.

For a server s, the maximum WCET of any clients in the client process between the request and the last reply is important when calculating the effect of preemptions, and is denoted $C_{s,client}$. This variable is also zero if there are no replies.

From a temporal point of view an event process behaves like a server to both the source and the target. Like with a server, a source that sends a trigger to an event may up to a certain point be preempted by another source of that event. The same applies to a target when it notifies the event that it is ready. When doing the schedulability analysis one can therefore treat an event as a server.

4.2. Schedulability Analysis of the System

A client can be preempted by multiple other clients and by each of these clients multiple times. As seen in the source code in Figure 6, a server cannot change clients while in phase two. Therefore, if the client is preempted in the second phase its work will be driven to complete with the deadline of the preempting task. In contrast, if a client is preempted in the first phase its work is in effect given away to the preempting task, and it will have to execute $C_{request}$ over again to make the server ready after the preempting process has finished. The worst case is that the client has executed *almost all* of $C_{request}$ every time it is preempted, in which case each preemption leads to another $C_{request}$ of work for the preempted client.

From the preempting client's point of view, preempting a task in its first phase in a server may save up to $C_{s,request}$ of execution, because of the work already done by the task being preempted. On the other hand, if preempting in the second phase, it will have to drive the current client to finish before the server becomes ready. This may also include execution of code in the process of the preempted client. The worst-case is that the current client has just begun the second phase, in which case preempting it leads to $C_{s,client} + C_{s,reply}$ of extra work for the preempting client.

For any two tasks A and B, where $D_A < D_B$, the deadline propagation graph can be used to identify the servers in the network where A will have to drive B if A preempts B. Such a server is called a *critical server* for the pair of tasks (A, B).

Definition 2 (Critical Server) *A critical server for the pair of tasks (A, B) is a server that can be accessed directly or indirectly by both A and B, but not only through the same server.*

If a server s_2 is accessible by two tasks only through server s_1, then preemption must necessarily happen in s_1, because both tasks cannot at the same time hold s_1, which is necessary to access s_2. In Figure 5, servers 1, 2 and 3 are critical servers for (A, B), while server 3 is the only critical server for both (A, C) and (B, C). The set of critical servers for any two tasks A and B is denoted $\mathrm{crit}\ (A, B)$.

If task A is to preempt task B, then A must start after B, and have an earlier absolute deadline. For tasks A and B, where $D_A < D_B$, this means that A can preempt B at most

$$\left\lfloor \frac{D_B}{D_A} \right\rfloor \tag{3}$$

times for each instance of B under EDF. In other words, Equation 3 is the number of consecutive instances of A that fits completely within a single instance of B. Note that a single instance of A can preempt B at most once.

$\widehat{C}_X$ is the execution time required by code local to X; the $\max$ function returns the largest numerical value of a set. T is the set of tasks. With these definitions, C_A can be written as:

$$C_A = \widehat{C}_A + \sum_{s \in \text{acc } A} (C_{s,request} + C_{s,reply}) + \sum_{X \in T, D_X < D_A} \left\lfloor \frac{D_A}{D_X} \right\rfloor \max_{s \in \text{crit } (A,X)} C_{s,request} + \sum_{X \in T, D_X > D_A} \max_{s \in \text{crit } (A,X)} (C_{s,client} + C_{s,reply}) \quad (4)$$

The first line in the equation is the amount of execution required by the task process itself, plus the execution required by mandatory accesses to other servers. The second line is sum of the worst-case penalties for being preempted by a given task, multiplied by Equation 3, which is the maximum number of preemptions by that task. The third line is the sum of the worst-case costs of preempting each other task that may be preempted. Equation 4 can be extended to include the scheduling overhead associated with preemptions, as the maximum number of preemptions can be found using Expression 3.

When the Cs of all the tasks have been found, then the system can be tested for schedulability with Equation 1.

5. Discussion

Schedulability analysis in systems of communicating processes is not well developed. This paper presents an algorithm for finding sufficient schedulability conditions for Toc programs. The analysis presented here is conservative, so dimensioning the processing power of a system based on this analysis may lead to low utilization. The analysis has not been tested on an actual system, and it remains to see just how conservative it is in practice. Nevertheless, this paper shows that, subject to certain restrictions, schedulability analysis is possible in principle for systems of deadline-driven communicating processes. Also, in some real-time applications, proof of schedulability is more important than a high utilization.

Sporadic tasks are common when programming in Toc, but rare in traditional real-time programming, where they are mostly used to model error-handling. The schedulability analysis in this paper is developed on top of the traditional model, where sporadic tasks are assumed to execute continuously with their relative deadlines as periods. This is a conservative assumption that is particularly unfortunate for systems largely based on sporadic tasks.

Many legal constructs in Toc programs lead to conflicts between timing requirements. This is for example the case when using an `ALT` within a `TIME`, using a single channel to communicate between tasks, or triggering sporadic tasks without an event-like formalism. It can be argued the language has far more expressive power than what can be safely used. Some of this is due to the language being syntactically based on occam, while being semantically very different. Indeed, the deadline-driven semantics of Toc and the strict round-robin semantics of occam may be too different to be supported by the same basic statements. However, the deadline-driven semantics of Toc is based on occam because occam is the existing language that best supports the concept of parallelism and synchronous communication needed by Toc. Extended rendezvous is another occam concept that has proven to be immensely useful in Toc; the use of extended rendezvous to manage deadlines cannot easily accomplished by other means. The clear separation of primitive and constructed processes that exist in occam is also the basis for Toc's definition of laziness, and it less clear how laziness can be defined in a C-like language, for example.

6. Conclusions and Future Work

This paper has introduced design patterns for communicating between tasks and triggering sporadic tasks. The patterns allow dependencies and conflicts between timing requirements to be avoided. However, whether they provide enough expressive power to design general systems or whether they scale well to larger systems is uncertain, as Toc is still an experimental language in an early stage. Composition of patterns with respect to timing requirements has also not been studied, and is a potentially interesting subject for future work.

When programming communicating systems with deadline propagation and laziness, all functionality of the system must be implemented with an explicit timing requirement. While the principle of timing requirements on all calculations is intuitively sound, it leads to the discovery of requirements that traditionally would have been unpronounced. Some of these requirements occur in the form of new sporadic tasks. Sporadic tasks therefore occur more frequently when programming in Toc than in traditional real-time programming, but their use leads to conservative schedulability analysis. Developing a framework for better specification and handling of sporadic tasks in schedulability analysis represents a potential for significant improvement.

The schedulability analysis presented here is based on the traditional approach, where the analysis is NP-hard unless $D = T$ for tasks where the relative start-times are not known. Because of the ability to trigger tasks in Toc, the relative start-times of tasks are generally not known. Being able to concisely specify tasks with shorter deadlines than period is a useful property of Toc, and the scheduling analysis approach would be better if it also supported this.

Consistent use of the design patterns to achieve decoupling of timing requirements limits the ways the language can be used. Toc has inherited occam's process statements almost unaltered, while having temporal semantics that are entirely different. A re-evaluation of Toc's statement model to make these design patterns embodied in the language may reduce the need for overly restrictive design rules.

The lack of a formal framework for analyzing Toc programs is also unfortunate. CSP can be used directly to analyze occam code, because CSP's principle of maximum progress maps to occam's strict round-robin scheduler. Although Toc is based on occam, the fact that scheduling is lazy rather than strict means that there is no direct mapping between Toc code and CSP. Developing a way to enable formal analysis of deadline-driven systems is an important objective for future work.

References

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

[2] Da Qing Zhang, Carlo Cecati, and Enzo Chiricozzi. Some practical issues of the transputer based real-time systems. In *Proceedings of the 1992 International Conference on Industrial Electronics, Control, Instrumentation and Automation*, pages 1403–1407. IEEE CNF, 1992.

[3] Peter H. Welch and Frederick R. M. Barnes. Communicating mobile processes: introducing occam-pi. In A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer-Verlag, apr 2005.

[4] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[5] A. William Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Europe, Hertfordshire, England, 1998.

[6] Martin Korsgaard and Sverre Hendseth. Combining EDF scheduling with occam using the Toc programming language. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2008*, pages 339–348. IOS Press, sept 2008.

[7] Martin Korsgaard, Sverre Hendseth, and Amund Skavhaug. Improving real-time software quality by direct specification of timing requirements. In *Proceedings of the 35th Euromicro SEAA Conference*. IEEE Computer Society, 2009.

[8] Peter H. Welch, George Justo, and Colin Willcock. High-level paradigms for deadlock-free high-performance systems. In *Transputer applications and systems '93: Proceedings of the 1993 World Transputer Congress 20-22 September 1993, Achen, Germany*, pages 981–1004. IOS Press, 1993.

[9] Jeremy M. R. Martin and Peter H. Welch. A design strategy for deadlock-free concurrent systems. *Transputer Communications*, 3(4), June 1997.

[10] D. B. Skillicorn, Jonathan M. D. Hill, and W. F. Mccoll. Questions and answers about BSP, November 1996. Oxford University Computing Laboratory.

[11] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[12] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement User Manual*, June 2005. Web page: www.fsel.com.

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

[14] Butler W. Lampson and David L. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, feb 1980.

[15] Liu Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep 1990.

[16] Theodore P. Baker. A stack-based resource allocation policy for realtime processes. *Proceedings of the 11th Real-Time Systems Symposium, 1990.*, pages 191–200, Dec 1990.

[17] Alan Burns and Andy J. Wellings. Synchronous sessions and fixed priority scheduling. *Journal of Systems Architecture*, 44:107–118, 1996.

[18] Sanjoy K. Baruah, Rodney R. Howell, and Louis E. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990.

[19] Reinhold Heckmann and Christian Ferdinand. Worst case execution time prediction by static program analysis. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004.*, page 125. IEEE CNF, Apr 2004.

[20] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold 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*, pages 21–24. IEEE Computer Society, 2005.

[21] Reinhard Wilhelm et al. The worst-case execution-time problem—overview of methods and survey of tools. *Transactions on Embedded Computing Sys.*, 7(3):1–53, 2008.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.