NTNU
Innovation and Creativity

Introducing Time Driven Programming using CSP/occam and WCET Estimates

Martin Korsgaard

Master of Science in Engineering Cybernetics
Submission date: August 2007
Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

Formulate a programming language that allows for explicit specification of timing requirements. Write a compiler and scheduler for the language. Discuss limitations and uses.

Assignment given: 07. March 2007
Supervisor: Sverre Hendseth, ITK

Summary

This thesis describes an experimental programming language called TIME/-occam. TIME/occam, like occam, is based on Communicating Sequential Processes (CSP), a branch of process algebra that allows computer programs to be modelled and verified mathematically and mechanically.

TIME/occam uses synchronous channel communication as the only legal communication between parallel threads, and prohibits shared variables. Simple statements allow programs to specify parallelism directly.

The main innovation in TIME/occam is the TIME statement, which allows the deadline and timing requirements of a task to be specified directly in the language. The TIME statement takes a block and a deadline, and tells the scheduler that the block must be completed within the deadline. It also states that the statement following the TIME block is not allowed to start until the timer expires. This can be used with a loop to create periodic tasks. Statements that have no timing requirements will never be executed. Because channel communication is synchronous, channels allow timing requirements to pass on from one task to another dependent task.

The use of channels and timing requirements allow execution of a program to be planned some time in advance. It is assumed that execution time estimates can be found on line, although no viable solution for this exists at this time. Planning execution opens up new possibilities for dealing with missed deadlines.

A compiler and scheduler have been implemented for the language. The scheduler is not complete; in particular it lacks the re-planning and execution algorithm.

The thesis also contains an introduction to real-time and concurrent programming, and describes some of the difficulties that arise from preemptive scheduling of dependent threads. There is also a discussion on worst-case execution time analysis and related hardware issues.

An example implementation of “dining philosophers” is presented, and it is explained how such a program is scheduled and executed in TIME/occam. Finally, the limitations of such a concept are discussed. In particular, there is a question whether or not a very heavy scheduler like the TIME/occam scheduler can be used in practice.

Contents

List of Figures

List of Tables

Part I

Programming with Time

Chapter 1

Introduction to Real-Time Systems

When browsing the web or typing a document in a word processor, we want our computer systems to be reasonably fast. It can be irritating if a web page loads slowly, or if it takes too long from typing a word until it appears on the screen. However, if the page finally loads or the word finally appears, we cannot say the computer has failed, only that it has been slow.

For a rocket control system this is not the case. If it reacts too slowly then the rocket may crash. Reacting too slowly may be no worse than not reacting at all, or reacting the wrong way. The same principle applies, less dramatically, to a digital music player. If the player uses too long time to load a sample, it will cause a discontinuity in music. That may sound just as bad as loading the wrong sample.

Such systems are called *Real-time systems*. They are characterized as systems, where not producing results on time is as much a failure as producing incorrect results. Most real-time systems are *embedded*, which means they are specialized systems built to do one particular job, unlike the multipurpose computers used at home or at work. Embedded systems are everywhere, controlling phones, dish-washers, cars and much more.

1.1 Real-time System Terminology

A Real-Time system may have several tasks, each with its own timing requirements. For instance, one task may be to control the system, by making sure the system responds correctly to external events. Another task may be to monitor the system health to look for errors or inconsistencies. A third task may be to provide feedback to operators.

It may not be possible to complete these tasks one at a time. For example, if there is a short, frequent task, and a long, infrequent task running on the same processor, then executing the long task without interruption may

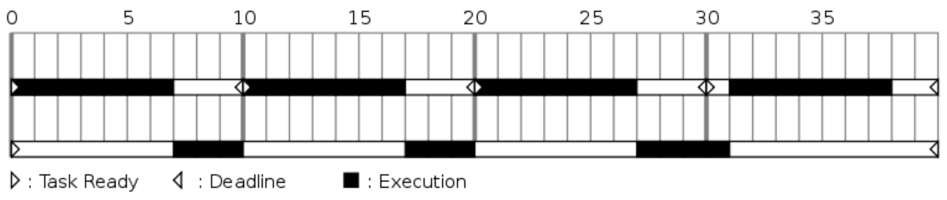

Figure 1.1: Example where scheduling is required

break the timing requirements of the shorter task.

The solution is to break each task into small segments that can be interleaved, so that they appear to be running in parallel. The tasks are then referred to as *processes* or *threads*. A *scheduler*, typically a component of the operating system, will then select which thread is to run at any given time. Bits of the long thread can be inserted in between execution of the short, frequent thread, so that both will satisfy their timing requirements.

An example is presented in figure 1.1. One periodic task has a deadline and period of 10 time units and an execution time of 7. In parallel, another task runs with a deadline at 40 and an execution time of 10. Without the latter task being split up, the shorter tasks could not possibly all complete before their deadlines.

1.1.1 Preemptiveness

If the scheduler can take control from one thread and give to another by force, it is called *pre-emptive*. Otherwise it is non-pre-emptive, and each thread has to voluntarily return control to the scheduler after running for some time.

Preemption has a number of advantages. It is typically achieved by adding scheduler code to a hardware timer interrupt. This is transparent to the threads themselves, so the designers of the threads do not need to interact with neither the scheduler nor the other threads in the system. This works well as long as the threads are independent.

If threads are independently designed, then pre-emption is the more stable and secure solution. In a non-pre-emptive system, each thread is dependent on the correctness of other threads. If a thread fails to return control to the scheduler, then all other threads will fail as well. All modern desktop multitasking operating systems therefore use pre-emptive schedulers, because they run independently designed applications.

Because pre-emption blocks threads using interrupts, threads may be pre-empted between any two processor instructions. There are times where such pauses of execution would lead to failure, for example if a process is sending a stream of data over some communication channel. A break in the data stream may cause the receiver to assume that the line is down.

Attribute	Symbol	Description
Task Arrival Time	A	The time of which the task arrives, or is detected or invoked.
Start Time	S	The time the task starts to execute. This is decided by the scheduler.
Ready Time	R	Earliest permitted start time.
Computation Time	C	The processing time needed to execute the task. On a system that supports processor frequency scaling, the computation time is specified for full speed.
Deadline	D	The latest possible time for the task to finish, used either in relative or absolute manner. Relative deadline is relative to ready time.

Table 1.1: List of task timing attributes.

Another example of a thread that should not be pre-empted is the scheduler itself. To avoid pre-emption, threads can temporarily disable interrupts. This requires hardware access, which is typically limited to the operating system and drivers.

1.1.2 Timing of Tasks

In real-time applications, threads have various timing requirements associated with their execution. The definitions that will be used here are the ones used in Nissanke [1]. They are listed in table 1.1.

Tasks may be classified as *periodic* or *sporadic*. A periodic task is a repeating task, with a fixed period between each ready time. A sporadic task is any irregular task, repeating or not.

Traditionally, a *background task* is a task that is only allowed to run when no non-background tasks need to.

1.1.3 Soft, Hard and Weakly Hard Tasks

A Real-time task, for which a miss of any one deadline represents a total failure, is called *hard*. Conversely, if some deadlines may be missed occasionally, it is called *soft*. A real-time task which may be occasionally skipped, but which results are useless after the deadline has expired, is called *firm*.

Computer Control Systems

Control systems are based on calculating an output based on a measurement. The formula used to calculate the output is typically continuous, so to use such a formula in a computer control system it must be sampled. Sampling may induce instability if done too infrequently, but the minimum acceptable sampling interval can be found mathematically as a function of the maximum frequencies in the system.

However, the mathematics usually assumes that the output is set instantly after the measurements are read. Computation times and disturbances from other processes mean that there will be delays and jitter (variations in sampling frequency). Systems are therefore always sampled more often than the absolute minimum frequency.

The concept of deadlines is awkward when used on these systems, as the relative deadline is assumed to be zero, or at least much less than the period, $D \ll T$. Note that if the computation time exceeds the period, then it will not only affect the validity of the next sample, but also of the next.

How "hard" is a computer control system? This depends on many things, including the degree of oversampling. If the system is massively oversampled, say by a factor of 2, then skipping a single sample will not affect the performance or stability of the system. Missing every 10th sample might not affect the system either, but missing two in a row may.

Multimedia Systems

Multimedia systems that play video or audio are other examples of real-time systems. Frames and samples must be ready at fixed rates. If a video player cannot draw a frame in time, and instead uses the previous frame, this is hardly noticeable to the user unless many frames in a row are skipped. The same applies to audio: Sparsely distributed single sample errors may be inaudible, but several in a row will reduce audio quality.

The concept of deadlines is clear in these cases: The deadline is when the frame needs to be displayed, or the audio sample played. In this case the deadline equals the period, $D = T$.

Weakly Hard Systems

To specify the timing requirements for these systems, it is not enough to say that for instance no more than 1 out of 100 deadlines can be missed, because 100 missed deadlines followed by 9900 deadlines met is far worse than missing every 100th deadline.

Bernat et. al [2] defines a *weakly hard* real-time system to be one in which the distribution of met and missed deadlines are precisely bounded. They define *strongly hard* systems as the special case in which there may be no missed deadlines.

They specify the following four weakly hard real-time constraints:

- For any m deadlines in a row, at least n must be met. This is denoted $\binom{n}{m}$, and reads "meets any n in m".

- For any m deadlines in a row, at least n consecutive deadlines must be met; denoted $\left\langle {n \atop m} \right\rangle$ and reads “meets row n in m”.
- For any m deadlines in a row, a maximum of n deadlines may be missed; denoted $\overline{\binom{n}{m}}$ and reads “misses any n in m”.
- For any m deadlines in a row, a maximum of n consecutive deadlines may be missed; denoted $\overline{\left\langle {n \atop m} \right\rangle}$ and reads “misses row n in m”.

Bernat et. al uses these definitions only on periodic tasks, and have more complex definitions for repeated tasks with irregular periods.

1.2 Scheduling of Independent Threads

The classical scheduling problem is this: Given a set of tasks with specified periods and computation times; how to interleave the execution of the tasks so that all possible deadlines are met? If there are deadlines that are not met, the schedule is said to *overflow*. Tasks are assumed to periodic and independent.

This section will explain two important scheduling principles that both assume independent threads: *Earliest deadline first* and *Rate-monotonic priority ordering.*

1.2.1 Earliest Deadline First

The most obvious scheduling strategy is Earliest Deadline First (EDF). Do first what hurries the most. This is a *dynamic* scheduling strategy, because it needs to compare the actual state of the running threads to know what to do.

The *processor utilization* is the average processor load used when scheduling the task set. It is denoted U, and can be found with the following formula:

$$U = \sum_i \frac{C_i}{T_i} \tag{1.1}$$

Naturally, a task set can never be scheduled if $U > 1$. EDF is an *optimal* strategy, which means that it is possible to schedule any task set that has $U \leq 1$. In other words: If it is possible to schedule a set of threads so that all the threads meet their deadline, then it is also possible to do so using EDF.

There are two main disadvantages of EDF. One is that it gives the programmer no control of what happens in case threads miss their deadlines. If something delays a task in a task set with a high utilization, the result may be a cascade of missing deadlines. The other main disadvantage of EDF is that it is hard to implement efficiently, as the scheduler needs to keep track of the deadline for all threads at any given moment.

Examples

A task set is given in table 1.2. Three periodic tasks exist in the system, and for convenience, they initially start at the same time. Here, the deadline is set equal to the periods. The processor utilization is $\frac{3}{6} + \frac{4}{9} + \frac{1}{36} \approx 0.97$, so this task set can be scheduled with EDF.

The first ready time of all tasks is usually set to zero. But because the tasks have different periods, over time there will be different offsets between the tasks' ready times. The *hyperperiod* is the least common divisor of all thread periods, and is the minimum period that contains all possible mutual offsets between threads.

In this task set, the hyperperiod is the period of the longest thread. A scheduling of this example is shown in figure 1.2. Another example is given in table 1.3 and figure 1.3.

1.2.2 Rate-monotonic Scheduling

Fixed Priority Scheduling (FPS) is a *static* scheduling technique, where each periodic task is given a priority, and where the scheduler always chooses the eligible thread with the highest priority to run. The main benefit of FPS scheduling is that it is simple to implement. Also, if deadlines are missed, they will be missed in an orderly manner, as the highest priority threads will always be executed first.

One such FPS scheme is rate-monotonic scheduling (RMS), or Rate-monotonic priority ordering (RMPO); where priorities are set according to task periods, so that the highest priority thread is the one with the shortest period. RMS assumes that the deadlines are equal to periods, that $D = T$. There is another similar scheduling method, deadline-monotonic priority ordering (DMPO), that works for $D < T$.

RMS cannot schedule all task sets that can be scheduled with EDF, but RMS is optimal in the sense that it can schedule any task set that can be scheduled by any other fixed priority scheme.

There is an elegant mathematical result related to RMS: Even though it cannot always schedule task sets with $U = 1$, it is guaranteed to schedule task sets where

$$U \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{1.2}$$

n is the number of threads in the system. This is only a lower bound, and higher utilization is usually achievable. This result is originally due to Liu and Leeland [3].

Task	S_i	C_i	T_i
1	0	3	6
2	0	4	9
3	0	1	36

Table 1.2: Task set example 1

Task	S_i	C_i	T_i
1	0	2	5
2	0	2	7
3	0	1	9

Table 1.3: Task set example 2

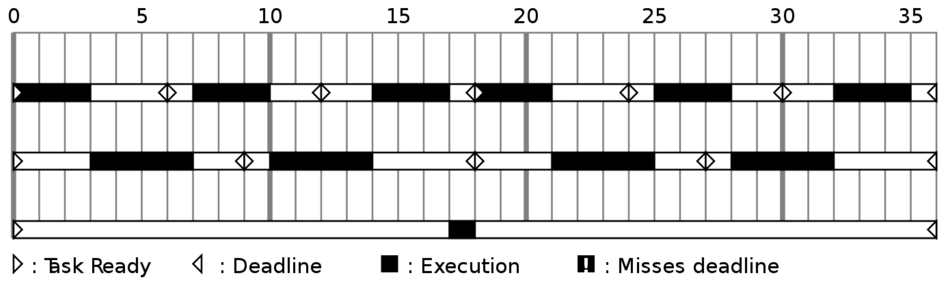

Figure 1.2: Example of EDF scheduling, task set 1

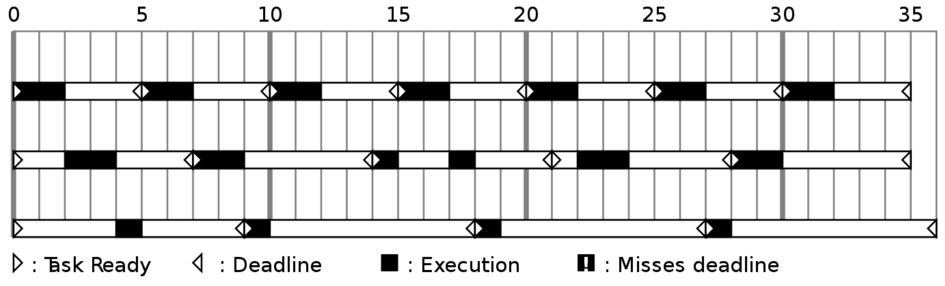

Figure 1.3: Example of EDF scheduling, task set 2

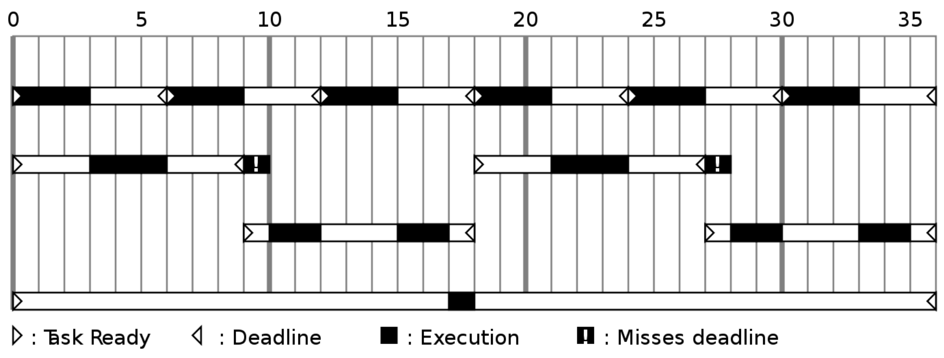

Figure 1.4: Example of RMS, task set 1

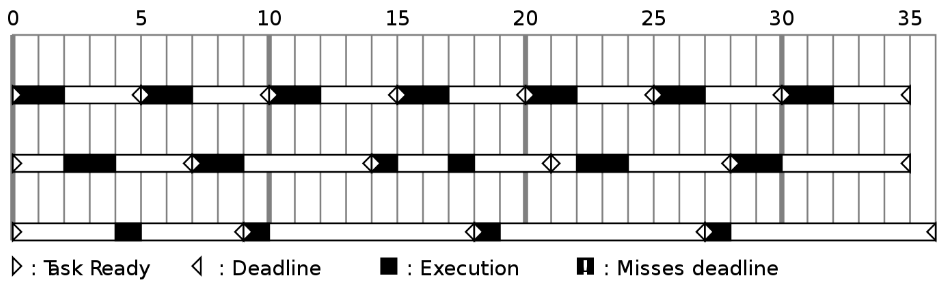

Figure 1.5: Example of RMS, task set 2.

For a large set of threads, this bound on processor utilization converges to ≈ 0.69. Although about 70% is less than the EDF bound of 100%, RMS is usually preferred over EDF. One reason is that one would never use a task set with 100% utilization even with EDF, because computation times are never accurate. Other reasons are the ease of implementing RMS, and the controlled way in which RMS scheduling fails.

A test on randomly generated task sets found that the average utilization limit for RMS is close to 88% (Lehoczky et. al [4]).

Examples

The task set in 1.2 has a utilization of 0.97. The formula in equation 1.2 yields a guaranteed lower bound of 0.78. It can therefore not guarantee for the scheduling of this task set. The scheduling fails, and is shown in figure 1.4.

The task set in 1.3 has a utilization of 0.80. It can still be scheduled as shown in figure 1.5, even though it fails the schedulability test. Notice how the second example is scheduled equally using EDF (figure 1.3).

1.2.3 Against Rate-monotonic Scheduling

Rate-monotonic scheduling is based on two often incorrect assumptions. The first is that the deadline is constant and equal to the period for all threads, $D = T$. The second is that threads are independent. These assumptions make for an elegant mathematical bound on processor usage: As long as the processor utilization is lower than 70%, any qualified task set is schedulable using RMS.

But deadlines are only sometimes equal to periods. Multimedia systems are examples where they are. Each frame must be ready before it is to be displayed, but does not need to be ready earlier. The frame rate is also constant.

Computer control systems, however, are examples of the opposite. Control theory assumes $D \ll T$ — that the time between the measurement and output is small. If such a system is scheduled using RMS, assumptions are not only broken, but contradicted, as RMS assumes $D = T$. Törngren [5] describes such a system, and notes that rate monotonic scheduling severely degrades control performance, because it allows too large deadlines in a motor servo loop. He also writes that the assumption $D = T$ is "postulated", rather than based on actual specifications.

The idea that a fixed priority scheduler behaves better during overflows because it is more predictable and allows more important threads to finish first (Lehoczky [4], Burns and Wellings [6], and more); is based on the assumption that the most important threads have the highest priorities. This makes sense by the original use of the word "priority", but does not automatically apply when priorities are set based on periods. Is it always so that the most frequent thread is the most important?

When $D = T$, fixed priorities for scheduling are best set using rate monotonic priority ordering, as is proved by many, including Nissanke [1]. A similar proof shows that when $D < T$, deadline monotonic priority ordering is the best choice. But a scheduling overflow should be considered an error condition, in which the important of tasks may not be dependent on periods or deadlines.

1.3 Dependent Threads

The processes of a system often need to communicate with each other, perhaps to exchange data or to signal a change of state for the system.

The most basic way of communication between threads, is to let the threads share variables. On simple systems this is easy to accomplish. If both threads know the memory address of a variable, they may read or write it as they please. On an operating system with memory protection the threads will need to share memory space, or else the variable must reside in explicitly shared memory.

```
/* Global */
int v = 1;

/* Thread 1 */          /* Thread 2 */
if (v > 0)              if (v > 0)
    v = v - 1;              v = v - 1;
```

Table 1.4: Example code for race condition

1.3.1 Race Conditions

Inappropriate use of shared memory in pre-emptive systems is one of the causes of *race conditions*. A race condition is a situation where the scheduling affects not only the timing of threads, but also their computational results.

Say the code in table 1.4 is executed in two simultaneously running threads on a pre-emptive scheduler, and that `var` refers to the same memory space in both threads. The two threads are not looping.

To analyze this rather small program, one must first find all the points of execution where the threads may be interrupted. Actions that cannot be interrupted are called *atomic*. From there one can draw a state diagram, containing all possible transitions and states that this program can be in.

The threads may be interrupted between any assembly instructions, of which there may be more than one at each line of code. The assembly instructions are machine dependent. We will assume that each thread runs the following pseudo machine code, in which every line is atomic.

```
if (v > 0) {
    t = v;
    t = t - 1;
    v = t;
}
```

The set of possible states for each thread is given in figure 1.6. The combined state space is given in figure 1.7. Note the explosion of states — 30 states — even for this limited program.

From a naïve look at the program, it may immediately seem like `var` will end up at 0 after the threads have finished, and that only one thread will be allowed to decrement. But this may not be the case, because each thread may be interrupted after the `if` statement, but before the decrement. If this happens then `var` may end up at -1. The sequences leading to `var = -1` can be traced in the state diagram. However, this result may be reversed by another race condition: If both threads read the variable before any of them get to write, both threads will decrement 1 and the end result will be

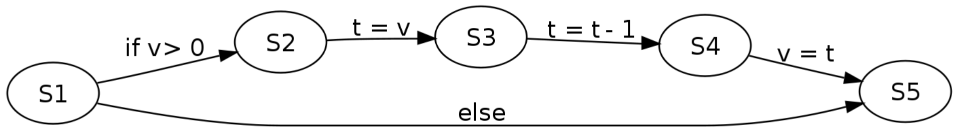

Figure 1.6: Thread states in race-condition example

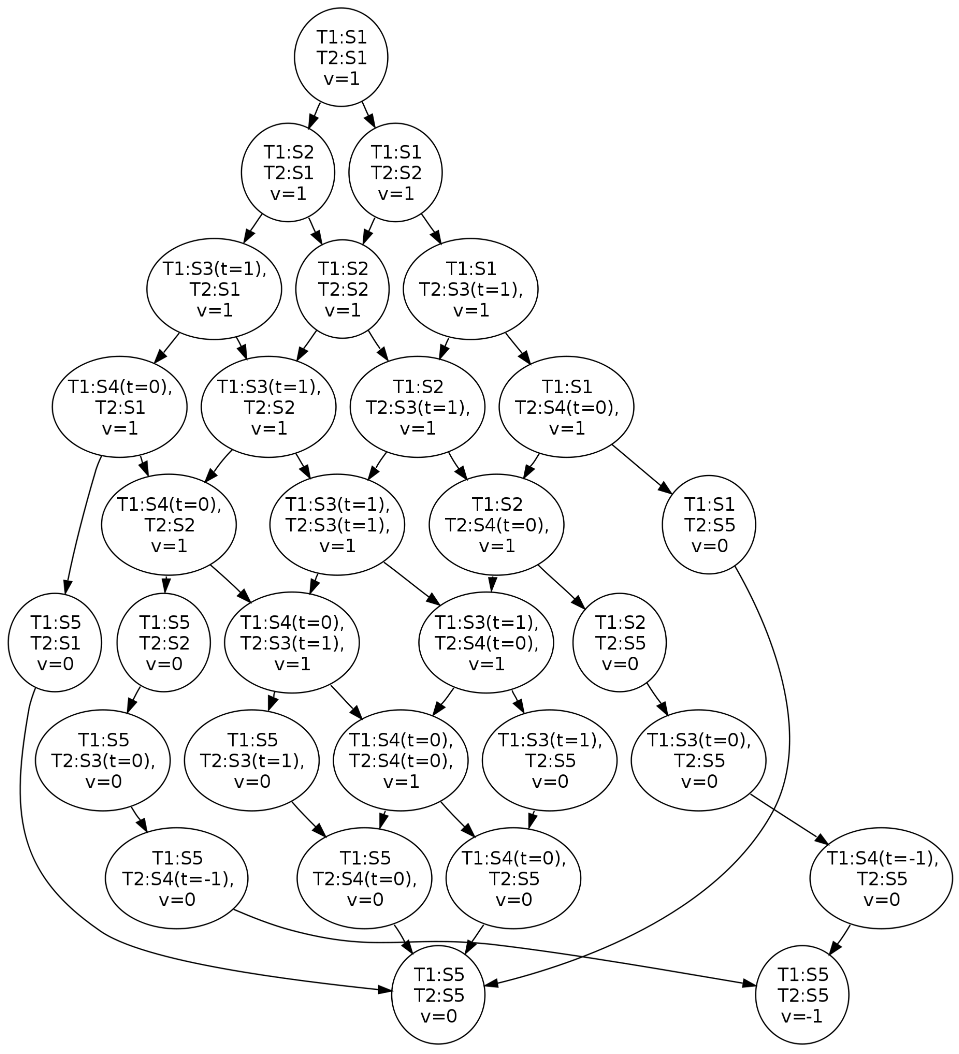

Figure 1.7: Combined state space in race-condition example

0. In the diagram, one can see the two ways to get to -1, and the four ways to get to 0.

Famous race conditions

Race conditions are a major concern in parallel systems, and have caused many serious problems. The worst incidents was due to one of the most famous computer bugs in history: The Therac 25 race condition. The Therac 25 was a radiation therapy machine that treated patients with cancer. Due to a race condition, unlucky timing of commands from the operating panel could make the machine produce high-power radiation when low-power was ordered. At least 5 people were killed. A report on the accidents [7] concluded that "Virtually all complex software can be made to behave in an unexpected fashion under some circumstances."

A race condition was also the cause of the Northeast American blackout of 2003. Due to a race condition, a data structure in a server at FirstEnergy was partially written by two different processes, and lost data integrity. The corrupted data structure was passed around to servers, which in practice stopped all servers from displaying alarms to the operators. An important alarm, due to a tree falling over a power-line went unnoticed, and escalated. In the end, several power plants had to shut down, and 50 million people lost power (Wikipedia [8]).

1.3.2 Mutual Exclusion

To avoid race conditions on shared variables, a common solution is to ensure that only one competing thread accesses a variable at a time. An "access" in this context means more than just read; in the example above the entire construct "test–read–modify–write" counts as one access.

Pieces of code that should not be interrupted by other code are called *critical regions*. One programming mechanism that allows such regions to be locked, is the *mutex*. A mutex is locked and unlocked. If the mutex is already locked when a thread attempts to lock it, the thread is blocked and cannot go further. When the original holder unlocks the mutex, the second thread may continue. Blocking and releasing threads is normally done by the OS. A *semaphor* is a counter that allows more than one holder before it blocks.

1.3.3 Priority Inversions

Use of mutual exclusion breaks priority assignments in scheduling. If a low-priority thread holds a mutex, then it cannot be pre-empted by a higher priority thread that will need the same mutex. Otherwise, the mutex will never be released, and the threads will deadlock. Forcing a release of the mutex, or ignoring it is not often a solution, as this will lead to the same race condition that made the mutex necessary to begin with. This deadlock is solved by letting the scheduler know about threads that are blocked waiting for mutexes, so that they will not be selected for execution. The lower priority thread is then allowed to continue running, until its mutexes are released. The situation of a thread waiting for a lower priority thread is called a *priority inversion*.

An *unbounded priority inversion* is caused by a chain of priority inversions. Say there are 10 threads. The highest priority thread (call it thread 1) is blocked waiting for the lowest priority thread (thread 10) to release its mutexes. This is priority inversion. Unbounded priority inversion occurs when thread 9 comes along and pre-empts thread 10. Now thread 1 waits for thread 10 which waits for thread 9. The chain may continue, and thread 1 — the highest priority thread — may end up waiting for all the other threads.

This problem almost led to the loss of NASA's "Mars PathFinder" [9]: A watchdog timer reset the system if a high priority data bus task had not run in some interval. A sequence of priority inversions blocked the data bus task for too long, causing sporadic resets. The problem was remotely debugged and eventually fixed.

1.3.4 Priority Inheritance and Priority Ceiling

Unbounded priority inversion can be fixed by *priority inheritance* or by employing the *priority ceiling protocol*.

Priority inheritance has a simple concept: When a high priority thread waits for a mutex owned by a lower priority thread, the lower-priority thread inherits the priority of the higher priority thread, which means that it will be allowed to finish without interruption from threads that normally would not be allowed to delay the high priority thread. This reduces the priority inversion.

But there is a catch. If a high priority thread is blocked waiting for a mutex owned by a lower priority thread it is not always enough to promote the low priority thread. The lower priority thread may also be blocked waiting for another thread, and this other thread must also be promoted. All threads in the chain must be allowed to finish before the high priority thread can execute, causing a possibly substantial priority inversion.

Priority ceiling is a scheduling mechanism, which gives priorities to resources depending on the highest priority thread that uses the resource. Whenever a thread locks a resource, it is promoted to that priority. Threads will not be allowed to run if they may try to lock a resource that is already locked. The method requires analysis of which threads that may lock which resources and is therefore more difficult to use.

1.3.5 Against Priority Inheritance

The title on this section is borrowed from Yodaiken [10], who wrote a light whitepaper arguing against the use of priority inheritance. He argues that one cannot find an upper bound to the delays caused by priority inversion, without static analysis of threads and resources. To those who argue that analyzing code is too difficult, he quotes a Barbie doll saying "Math is too hard". (He does not state who it is that means static analysis is too hard, and Doug Locke accuses him of a straw man attack in his rebuttal of the whitepaper [11]. Locke also argues that, although not perfect and possible to use incorrectly, priority inheritance is a useful tool.)

Yodaiken has some valid arguments against priority inheritance. The first is that it does not handle nested locks well. It prevents a thread getting scheduled if it would indirectly pre-empt another thread of higher priority, but if a higher priority thread is to get scheduled, it may still have to wait for a chain of other threads to finish, if those threads had already started.

He also notes that priority inheritance does not mix well with non-inheritable operations such as pipes and other blocking I/O operations, that implementing the algorithm adds complexity to the operating system, and that executing it will add further delays to the inversion.

1.4 Concurrent Programming

Mutual exclusion can be used to guarantee that critical regions are only accessed by one thread at a time. But the use of mutual exclusion leads to a number of other challenges.

1.4.1 Deadlock

Deadlocks occur when using multiple locks in a circular fashion, so that, for instance thread 1 is blocked waiting for thread 2 and thread 2 is blocked waiting for thread 1. The simplest case of this occurring is when threads lock locks in a different order, as in the example in table 1.5. The shortest path to deadlock is this: Thread 1 locks mutex 1, and is then interrupted. Thread 2 starts and locks thread 2. Now thread 1 is blocked waiting for mutex 2, and thread 2 is blocked waiting for mutex 1.

There are four necessary conditions for a deadlock, known as the *Coffman conditions.*

Mutual Exclusion: There exist resources which cannot be shared, and which may cause a thread to have to wait for another.

Hold and wait: A thread may wait for a resource, even if it has already locked another.

```
mutex m1, m2;

/* Thread 1 */           /* Thread 2 */
lock(m1);                lock(m2);
lock(m2);                lock(m1);
do_something();          do_something_else();
unlock(m2);              unlock(m2);
unlock(m1);              unlock(m1);
```

Table 1.5: Example code for deadlock

No pre-emption of resources: The holder of a resource is the only thread allowed to release it.

Circular wait: It may occur that a closed circle of threads is waiting for one another.

These are necessary, but not sufficient conditions; they do not guarantee that a deadlock occurs, but if one of them is not satisfied then there cannot be a deadlock.

The simplest way to avoid a deadlock is to remove circular waits, for instance by demanding that all threads lock their resources in the same order.

1.4.2 Starvation

Starvation is a more subtle concept, where some threads may run less frequently than necessary or expected. Both deadlock and starvation is well illustrated by the *dining philosophers problem*. The problem is illustrated in figure 1.8. A number of philosophers — usually 5 — sit around a table thinking and eating spaghetti. For reasons not very well accounted for they only have one fork per person, but they all need two forks to eat. The problem illustrates the common computing problem of resource contention.

The philosophers pick up one fork at a time, then eat for some time, then put their forks down and think for a while, before repeating the process. If a rule is made, that all philosophers pick up their left fork before their right, deadlock may occur, as all philosophers sit there holding one fork, waiting for the next.

If one philosopher is told to reverse this, and pick up the right fork before the left, then there can be no deadlock. There may, however, be starvation (literally, which was the idea behind the dining metaphor). Starvation may occur in many ways. If for instance two philosophers think more than the others, then they will get to eat less.

The deadlock solution of reverse fork order may also lead to starvation because one fork will then be more contended than the others. The left-first

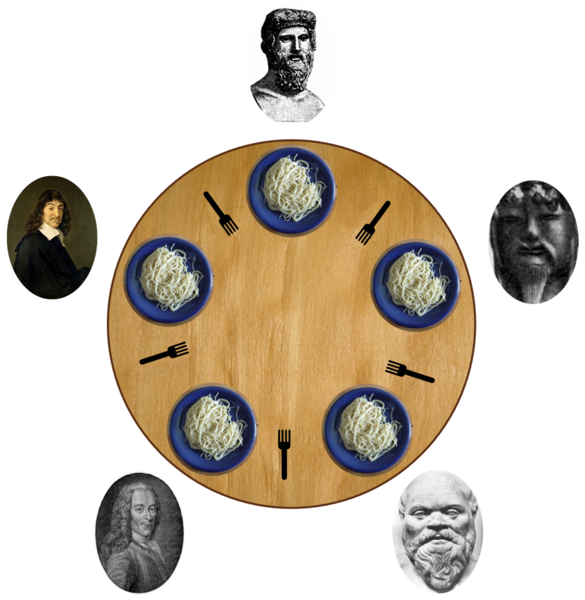

Figure 1.8: Dining philosophers illustration (from Wikipedia).

philosopher and his left neighbour will always have to compete for both forks, while the others only have to compete for both forks if both neighbours are eating.

1.4.3 Monitors

A mutex or semaphore in itself is a low-level programming structure that is easy to use incorrectly. There is no technical requirement that mutexes must be used to protect critical regions, or that it is the locker that must unlock a mutex. Unstructured use of mutexes leads to deadlocks and race conditions that can be hard to track down.

Monitors were designed to alleviate some of these problems. One of the primary goals of the monitor is to encapsulate critical regions and associated locking mechanisms into an object. The basic idea is to protect a variable so that it can only be accessed through certain procedures, and that those procedures guarantee mutual exclusion.

Java supports the building of monitors through the keyword `synchronized`; only one synchronized procedure for each object may be run at any time. Subsequent callers must wait until the ongoing call is finished. An example of a counter that supports atomic "decrease–if–not–zero" is given in figure 1.9. This implementation is clean, readable and to the point: Two threads cannot call neither of `inc()` nor `decIfNotZero()` at the same time.

A common situation is that a thread will have to wait for another thread to complete some task, while in a critical region. Java supports this with

```
class Counter
{
    protected int counter;

    public synchronized void inc() {
        counter = counter + 1;
    }

    public synchronized void decIfNotZero() {
        if (counter > 0)
            counter = counter - 1;
    }

    public int read() {
        return counter;
    }
}
```

Figure 1.9: Java monitor that supports atomic "decrease–if–not–zero". Initialization not included.

the functions `wait()`, `notify()` and `notifyAll()`. Waiting puts the thread to sleep, and releases the mutex lock, allowing other threads to enter synchronized functions. `notify()` wakes up one thread, if any threads sleep. Otherwise it does nothing. `notifyAll()` wakes up all threads, if any threads sleep.

One example is a bounded buffer that supports `get()` and `put()` operations. If `put()` is called when the buffer is empty, it blocks the calling thread until another thread calls put, in which case the first thread wakes up. Trying to put when the buffer is full has the same effect. The code is listed in figure 1.10. The code is still relatively clean and readable.

1.4.4 Nested Monitor Problem and the Inheritance Anomaly

Say the bounded buffer was implemented using the atomic counter. The counter is modified to lock if trying to decrease zero or to increase beyond max. The `get()` function of the buffer could then seem to be written like in figure 1.11. Note the minimal get and put functions. This bounded buffer does not work at all. The cause is known as the *nested monitor problem*: If a thread is blocked and waiting, because the buffer is empty, then it is waiting in the `Counter2` monitor. The `wait()` call only releases the lock on `Counter2`, so the caller still holds the lock on `Buffer2`. Because the buffer is locked, no calls to `put()` will be able to execute, and the system deadlocks. The best solution is never to use nested monitors.

Another problem with concurrent Java programming, is the `notify()`

```
class Buffer
{
    protected int counter;
    protecetd int data[];

    public synchronized int get() {
        if (counter == 0)
            wait();
        notify();
        return data[--counter];
    }

    public synchronized void put(int x) {
        if (counter == length)
            wait();
        notify();
        data[counter++] = x;
    }
}
```

Figure 1.10: Java bounded buffer. Initialization not included.

function. Because only one thread wakes up, this function only works if there is only a single reason that any threads may be waiting. This leads to another common monitor programming error, called the *inheritance anomaly.* The example is from “Real-Time Systems and Programming Languages [6].

Say that the bounded buffer is superclassed with functions allowing locking and unlocking the entire buffer. These functions must be synchronized with the buffer. If a thread wants to lock the buffer when the buffer is already locked, the thread will have to wait. When a call to `lock()` returns, the thread should be guaranteed exclusive access to the buffer. The buffer base class is the working version in figure 1.10. The code for the lockable buffer superclass is in figure 1.12.

The problem this time is that the calls to `notify()` may wake up the wrong thread. This may happen in a number of ways. For instance, say a thread is locked on a `get()`, because the buffer is empty. Then access to the buffer is locked. A call to `unlock()` may then release the first thread, even though nothing has filled up in the buffer. The counter would be decremented below zero, and the program will very likely crash.

To solve these problems in Java, there is a general rule of always using `notifyAll()`, and always looping around a conditional call to `wait()`. Now, if a thread is woken up that still has reason to sleep, it will go back to sleep again. Because all threads are woken up, there is no danger that only wrong threads wake up. This solution is listed in figure 1.13.

However, the solution is still buggy. Take the following scenario:

```
// Defective Code!
class Counter2
{
    protected int counter;
    protected int max;

    public synchronized void incIfNotMax() {
        if (counter == max)
            wait();
        counter++;
        notify();
    }

    public synchronized void decIfNotZero() {
        if (counter == 0)
            wait();
        counter--;
        notify();
    }
}

class Buffer2
{
    protected Counter2 counter;
    protected int data[];

    public synchronized int get() {
        return data[counter.decIfNotZero()];
    }

    public synchronized void put(int x) {
        data[counter.incIfNotMax()-1] - x
    }
}
```

Figure 1.11: Defective Java bounded buffer.

1. The buffer is full, and a producer thread tries to put something in it. The producer is blocked and must wait. It waits in the `Buffer.put()` function.

2. Now two locking threads tries to lock the buffer, which means that the last locking thread is blocked, and has to wait until the first unlocks. The last locking thread waits in the `LockableBuffer.lock()` function.

3. Then a consumer thread try to read from the buffer, but it is locked, and the thread must wait. It waits in the `Buffer.get()` function.

4. The first locking thread unlocks the buffer, and wakes all the threads up. The order of execution on threads that are woken up is random.

5. Say the first to wake up is the consumer, from the `LockableBuffer.get()` function. It takes an item through `Buffer.get()`, and returns.

6. Then the locking thread runs, and locks the buffer. It returns.

7. Then the first producer thread wakes up, *but it wakes up in* `Buffer.put()`*, and is not blocked, even if the buffer is now locked.* The producer puts an item in the buffer. The buffer is changed even though it is locked, contrary to specifications.

These examples illustrate some of the problems of concurrent programming. As usual, it is possible to fix this last bug also, the simplest way being not to use inheritance. However, more serious is the way in which the examples show that Java's object oriented paradigm does not work with Java's concurrent programming utilities. The principle of object-oriented design is that each object is responsible for its own inner workings, so that errors in one object cannot pass on to another. It should be possible to test a system part by part. The examples above show that this is not always the case. Neither nesting nor inheritance can be combined in concurrent programming without testing the system as a whole.

```
// Defective Code!
class LockableBuffer extends Buffer
{
    protected boolean locked;

    public synchronized void lock() {
        if (locked)
            wait();
        locked = true;
    }

    public synchronized void unlock() {
        if (!locked) throw new UnlockError();
        locked = false;
        notify();
    }

    public synchronized int get() {
        if (locked)
            wait();
        return super.get();
    }

    public synchronized int put(int x) {
        if (locked)
            wait();
        super.set(x);
    }
}
```

Figure 1.12: Defective Java lockable bounded buffer.

```
// Defective Code!
class Buffer
{
    protected int counter;
    protected int data[];

    public synchronized int get() {
        while (counter == 0)
            wait();
        notifyAll();
        return data[--counter];
    }

    public synchronized void put(int x) {
        while (counter == length)
            wait();
        notifyAll();
        data[counter++] = x;
    }
}

class LockableBuffer extends Buffer
{
    protected boolean locked;

    public synchronized void lock() {
        while (locked)
            wait();
        locked = true;
    }

    public synchronized void unlock() {
        if (!locked) throw new UnlockError();
        locked = false;
        notifyAll();
    }

    public synchronized int get() {
        while (locked)
            wait();
        return super.get();
    }

    public synchronized int put(int x) {
        while (locked)
            wait();
        super.set(x);
    }
}
```

Figure 1.13: Still defective Java lockable bounded buffer.

1.5 Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a branch of process algebra. It can be used to model parallel systems, and to test for certain properties. In particular, it can be used to model complex computer programs to check for deadlocks, race conditions, starvations and other bugs. A commercial tool that does this is Failure-Divergence Refinement (FDR), by Formal Systems. There is also a free tool available, LTSA, which is limited to finite systems. LTSA was made to accompany the book "Concurrency, State Models and Java Programming" by Magee and Kramer [12], and can be found on websites associated with that book.

1.5.1 States and Events

The basic token in CSP is the *process*. Processes are defined in Schneider [13] as:

> ...independent self-contained entities with particular interfaces through which they interact with the environment. This viewpoint is compositional, in the sense that if two processes are combined to form a larger system, that system is again a self-contained entity with a particular interface — a (larger) process.

Events change one process into another. One example is that of a fork, which may be picked up, and then put down, but not picked up twice without being put down first. This process is described in CSP as

$$FORK = pickup \rightarrow putdown \rightarrow FORK \tag{1.3}$$

The process $FORK$ can be viewed as a state machine with two states: One in which it may be picked up, and one in which it may be put down.

1.5.2 Choice

A philosopher can be modelled as a process which initially may think, then pick up either the left or the right fork, then pick up the other fork, then eat, and then think some more. The philosopher has a *choice* of which fork to pick up first. There are two types of choices in CSP. Internal choices are determined by the process to which they apply, and are denoted with a cap ($\sqcap$). External choice is chosen by someone else, and appears random to the process to which the choice applies. It is denoted with a box ($\Box$). The philosopher can choose which fork to pick up, and is therefore described the following way in CSP:

$$\begin{aligned} PHIL &= pickleft \rightarrow pickright \rightarrow eat \rightarrow putdown \rightarrow PHIL \\ &\sqcap\ pickright \rightarrow pickleft \rightarrow eat \rightarrow putdown \rightarrow PHIL \end{aligned} \tag{1.4}$$

1.5.3 Concurrency

Parallel processes may be formed by using double vertical pipes. (||). If any events are shared, they must be engaged in simultaneously by both processes. Modelling two processes, a mutex, and a critical region may be done as follows:

$$\begin{array}{rcl} MUTEX & = & p1.lock \rightarrow p1.unlock \rightarrow MUTEX \\ & \Box & p2.lock \rightarrow p2.unlock \rightarrow MUTEX \\ REGION & = & p1.read \rightarrow p1.modify \rightarrow p1.write \\ & \Box & p2.read \rightarrow p2.modify \rightarrow p2.write \\ P1 & = & p1.lock \rightarrow p1.read \rightarrow p1.modify \rightarrow p1.write \rightarrow STOP \\ P2 & = & p2.lock \rightarrow p2.read \rightarrow p2.modify \rightarrow p2.write \rightarrow STOP \\ SYSTEM & = & MUTEX||REGION||P1||P2 \end{array} \tag{1.5}$$

Note that the events that are not the same must have different names. When P1 locks the mutex, that is the same event as the mutex being locked by P1, but it is not the same event as P2 locking the mutex, so they must have different names.

In this process, "read – modify – write" must happen without interruption, because of the mutex. A sequence like

$$\begin{array}{l} p1.lock \rightarrow p1.read \rightarrow p1.modify \rightarrow p2.lock \rightarrow p2.read \\ \rightarrow p2.modify \rightarrow p2.write \rightarrow p2.unlock \rightarrow p1.write \rightarrow p1.unlock \end{array} \tag{1.6}$$

cannot happen because the mutex is not willing to participate in p2.lock between the actions p1.lock and p1.unlock. If a process like the mutex can sometimes engage in an event, like p2.lock, it must participate in all occurrences of the p2.lock events. A process that can sometimes engage in an event is formally said to have that event in its alphabet.

1.5.4 Send and Receive

Values can be passed on from one process to another, with a special form of event. Say process P1 and P2 can engage in the events $x.1 \ldots x.100$. By engaging in event $x.42$, process P1 can indirectly transfer the value 42 to process P2. The x is called a *channel*. The exclamation mark is used for sending, so the process can be described in CSP as:

$$P1 = x!42 \rightarrow STOP \tag{1.7}$$

Receiving is done denoted by a question mark. A process P2, which accepts a value along channel x, and then prints it, can be written as:

$$P2 = x?value \rightarrow \text{print}\,(value) \rightarrow STOP \tag{1.8}$$

1.5.5 Verification by Traces

With a program in CSP, one can specify certain attributes that one wants the program to comply with, and then run a test to verify that the program actually satisfies those attributes. For instance, one may wish to test that a critical region is indeed only accessed by one thread at a time. One way to specify those attributes is through trace properties.

A trace of a process is a set of events the process can engage in, from the beginning, but not necessarily until the end of the process. The trace set of a process is the set of all traces for a process. The trace set is not always finite. The trace set for the *FORK* process in equation 1.3 has the trace set

$$\begin{aligned}\mathrm{TRACE}\,(FORK) \;=\; &\{\langle\rangle, \langle pickup\rangle, \langle pickup, putdown\rangle,\\ &\langle pickup, putdown, pickup\rangle,\\ &\langle pickup, putdown, pickup, putdown\rangle, \quad \ldots\}\end{aligned} \tag{1.9}$$

CSP contains a rich set of ways to specify requirement for such traces. For example, one may want to check for the process set in equation 1.5 that there are no p2.read or p2.write between p1.lock and p1.unlock. This can be specified with the trace property:

$$\begin{aligned}P\,(tr) \;=\; &(tr = tr_{before}{}^\frown\langle p1.lock\rangle{}^\frown tr_{critical}{}^\frown\langle p1.unlock\rangle{}^\frown tr_{after})\\ &\Rightarrow tr_{critical} \downarrow \{p2.read, p2.write\} = 0\end{aligned} \tag{1.10}$$

This reads as follows: All ways to split the trace tr into part tr_{before}, followed ($^\frown$) by p1.lock, followed by $tr_{critical}$, followed by p1.unlock, followed by tr_{after}; must satisfy the requirement that the number of $p2.read$ or $p2.write$ events in the trace set $tr_{critical}$ must be zero.

There are a large number of other ways to specify this trace set requirement. Once a property like this has been established, there exists mechanical ways to check whether or not the property holds for a given CSP program. However, it is not simple to formulate a set of properties that catches all concurrent programming errors in a program. In particular, it may be hard to specify properties without knowing in advance which error to look for.

1.6 Occam and CSP programming

Occam was made to provide a suitable programming language for the INMOS transputer. Transputers were designed to be highly parallel, and no programming language at the time supported this natively. Occam is based on CSP.

Although occam became less popular as the transputer became less popular, it is still used, and compilers exist for other targets.

Occam is conceptually different from other programming languages in the way it handles parallel threads and communication between these. First, occam has native support for parallelism, using the `PAR` keyword. The `PAR` keyword takes a set of statements and executes them in parallel, for example:

```
PAR
  x := 2
  y := 4
  z := 6
```

Furthermore, it is not allowed to list a set of statements without specifying if they are to be executed in sequence or in parallel. The keyword for sequenced execution is `SEQ`.

Sharing a variable between parallel threads is illegal in occam. Communication between parallel processes must be done using channels. A channel must have one sender and one receiver. Sending and receiving is synchronous, which means that the sender must wait for the receiver to accept before it can proceed, and the receiver must wait for the sender to send. Channel communications can therefore be used as synchronization primitives. Sending is done using the "`!`" operator, and receiving is done using "?". The following example assigns the value 8 to the variable `z`:

```
PAR
  ch ! 8
  z ? ch
```

Originally, channels were also the way in which parallel transputers exchanged information. It was transparent to the programmer whether or not the communication was between different transputers.

The final conceptually fundamental statement is `ALT`, which chooses between several channels for input. The following example is a server that accepts `read.now` commands and then sends the value through the `read.value` channel. ("`.`" is a valid character in variable names). The server can also accept `update.now` commands, after which it sends a value over `update.get` and blocks other calls until it receives an updated value from the `update.set` channel. All channels are arrays, so that several parallel threads may use the server. This server protects a global variable, so that all read-update-write accesses become atomic.

```
INT value:
INT dummy:
ALT
  ALT FOR i=0 TO SIZE read.now
    read.now[i] ? dummy
      read.value[i] ! value
```

```
PROC buffer(CHAN OF INT in,out)
  VAL INT buffersize IS 5:
  [buffersize] CHAN OF INT buffer:
  PAR
    INT v:
    WHILE TRUE
      SEQ
        in ? v
        buffer[0] ! v
    PAR i=0 FOR buffersize - 1
      INT v:
      WHILE TRUE
        SEQ
          buffer[i] ? v
          buffer[i+1] ! v
    INT v:
    WHILE TRUE
      SEQ
        buffer[buffersize-1] ? v
        out ! v
:
```

Figure 1.14: Occam bounded buffer.

```
ALT FOR i=0 TO SIZE update.now
  update.now[i] ? dummy
    SEQ
      update.get[i] ! value
      update.set[i] ? value
```

1.6.1 Bounded Buffer Example

An implementation of a bounded buffer in occam can be fundamentally different from one in Java. The version presented in figure 1.14 uses a sequence of channels to store the values, there is no buffer array of INTs, but a buffer array of channels of INTs. The upper `WHILE` block reads from input and attempts to put what it reads into `buffer[0]`. The centre `WHILE` is replicated in parallel, and reads from `buffer[i]` and writes to `buffer[i+1]`, effectively pushing items through the buffer. The last `WHILE` loop writes to output. As long as there are spaces in the buffer, the middle loop will push items away from `buffer[0]`, which means that the first loop will be able to read more items. When the buffer is full, the middle while loop stops, and the first while loop will stop at "`buffer[0] ! v`", and be unable to read more items from the `in` channel.

1.6.2 CCSP and JCSP

Some programmers argue that the synchronous occam approach to concurrent programming is inherently more stable and secure than using mutexes or monitors. Because it may be too ambitious for companies to leave the old habits of C/C++ and Java, CSP extensions have been added to those languages. CCSP is CSP with C-like statements, compiled with a separate CCSP compiler. JCSP is a library to use with Java, but does not change the underlying language. Teig [14] claims to have great success in using these principles, and states that "Concurrent programming doesn't have to remain one of the most difficult areas in software design."

1.7 Dynamic Voltage Scaling

One interesting new application of scheduling is dynamic voltage scaling, or DVS. DVS is a method for reducing energy consumption by lowering the frequency and voltage of the processor, when less computational power is required.

1.7.1 Physical Background

Every transistor switch in a processor uses energy. For electronic reasons that are well explained in Hong et al. [15], the power use is roughly proportional to the square of the processor voltage times the frequency. The last proportionality is natural, because as the frequency increases, the number of switches per time unit increases with it.

$$P \propto f \times v^2 \tag{1.11}$$

The frequency of a processor is often scalable. Voltage is sometimes also variable, but higher frequencies demand higher voltages. Only if the frequency is reduced, may the voltage be reduced. The maximum voltage of a processor is typically 1-2V. The minimum voltage is determined by noise sensitive circuits in the processor, and is typically about half to two thirds of maximum voltage. Reducing the CPU voltage with 33% reduces the power consumption by half, halving the CPU voltage reduces the power consumption by 75%.

There is potential for lowering the minimum voltage in many processors, because voltage scaling functionality is usually tacked on, rather than being part of the original design of the processor (Zhai et al. [16]). CMOS circuits are generally able to operate with voltages of about 200mV.

1.7.2 Slow is Optimal

Much energy is lost due to simple clock propagation, and a common way to conserve energy is to turn off parts of the processor that is not currently in use, by disabling the clock. For a task set with utilization U, running on a processor with power usage P, the average power saved by sleeping is at most:

$$\frac{P_{1-U}}{P_1} = U \tag{1.12}$$

If $U = 0.50$, then the power consumption can be as low as 50% of full power. Instead of executing fast and then sleeping, execution can be done slowly, stretching out to fill timing requirements.

$$\frac{P_{1-U}}{P_1} = U \times \left(\frac{v_{1-U}}{v_1}\right)^2 \tag{1.13}$$

An example from Snowdon et al. [17], allows for 25% lower voltage at half of the processing frequency. This reduces the power consumption to 32% of full power.

In larger systems, for instance systems with hard drives, power consumption may not dominated by the CPU, and DVS will be relatively less effective than for smaller systems. Snowdon et al. also points out many obstacles of DVS. For instance, memory and bus frequencies must also be lowered. Also, processors have considerable static power consumptions that are independent of frequency, further reducing the effectiveness of DVS.

Chapter 2

Execution Time Analysis and Hardware

Execution Time Analysis is the problem of estimating how much time a program or task needs to run on a specific machine. This information is necessary in real-time systems scheduling, without it no analysis of schedulability would be possible. Estimating execution time is nevertheless quite difficult. One reason is that the time spent on a computation depends on exactly what is being computed, which may vary from execution to execution.

Another reason is that modern computer architectures feature complex mechanisms in order to increase speed and power efficiency. Many of these mechanisms are focused on improving the average speed, and often lead to greater differences between best-case and worst-case execution times of programs. The effects of each of these improvements are generally difficult to predict, and they interact in such a way that they cannot be analyzed separately.

If deadlines are to be guaranteed met, then execution time analysis is necessary no matter how tasks are being scheduled. However, in simple earliest deadline first or fixed priority scheduling, knowing execution times is not necessary to actually run the system, only to prove that it will work.

This chapter will assume detailed knowledge of cache, processor pipelines and branch prediction. A good reference on the subject is the book "Computer Architecture: A Quantitative Approach" by Hennessy [18]. The chapter is an excerpt from an earlier paper on execution time analysis and hardware [19].

2.1 WCET Analysis

The execution time of a program often depends on the input data given to that program. It may also be affected by external factors or interference.

In the context of real-time systems it is therefore common to examine a program's *worst-case execution time*, or WCET, which is an upper bound to the execution time.

WCET Analysis is often divided into *Static* and *Measurement-based.* Static execution time analysis concerns determining the execution time of a program prior to running the program. Measurement-based WCET analysis is based on interpreting execution times of actual runs of the program. The distinction between the two is often more theoretical than practical - as static analysis often use simulation results, which are not fundamentally different from measurements.

A WCET estimate is said to be *valid* if it is a guaranteed upper bound to the execution time. To be of any use, the WCET estimate must also be *tight*, meaning that it is close to the actual WCET of the program.

WCET Analysis can be divided into two interdependent aspects (Kirner et. al [20]):

Execution Path Analysis This aspect concerns the program's control flow, particularly finding which path through the program's control statements that take the longest time to execute. The focus of this aspect is flow control, and tracking feasible execution paths.

Execution Time Analysis Execution time analysis tries to find the exact time a given execution path takes to execute. The focus here is on how a given stream of instructions would be executed hardware.

2.2 Execution Path Analysis

One of the key problems of static WCET analysis is determining a program's *execution path.* The execution path is the unwrapped sequence of instructions a program executes, and reflects how the program manoeuvres through loops and conditional statements (control statements).

The execution path of a program often depends on the program's arguments (input data), which means that a program may have many execution paths. Of all possible execution paths of a program, the path that takes the most time to execute is known as the *worst-case execution path*, or WCEP.

It is not a simple task to determine the execution path of a program with given arguments without running it. A program may for instance have loops with conditional exit statements, where the loop condition is a complex function of several variables. Use of recursion also makes finding the actual execution path difficult.

Pragmatically, it is useful to distinguish between programs where the execution path is only dependent on the size of the input data, and programs where the execution path is dependent on both the size and the value of the input data. The latter is said to have *data-dependent branches.*

This distinction is useful, because the size of input data is often fixed and known in advance, whereas the value of data is not. In real-time systems many programs, such as video compression/decompression algorithms, digital filters and regulators, work on pre-known and constant data sizes. If such a program does not have data-dependent branches this will greatly simplify finding the WCET for that program.

More often than not, however, programs do have data-dependent branches. It may still be possible to determine the WCEP of the program from its structure, but it may also be necessary to check various legal sets of input data to find the WCEP. Computerized tools can be used to extract such information, and if the conditional expressions are relatively simple, then such a tool may work well. But if the conditional statements are tangled or are functions of previously calculated data, then the complexity of finding the WCEP may be too high.

2.2.1 WCEP of Linear vs. Binary Search

As an example, consider two forms of search algorithms. The first, linear search, takes any list of numbers and searches for a given value in that list:

```
function LinearSearch(list[], value)
    for i = 1..length(list) do
        if list[i] == value
            return i
        end if
    end for
    return 0
end function
```

For this algorithm, the WCEP is encountered when the value is not found in the list. Then, `length(list)` iterations of the loop will be executed. It is easy to see, and for a computerized tool to find, that no more than `length(list)` iterations can possibly occur, because the loop has simple bounds.

This is not the case for the binary search. The binary search algorithm takes an ordered list of numbers and searches for a given value. This is done by checking the middle value; if the middle value is lower than the value being searched for, then search above it, otherwise search below it. This halving of the search range is done recursively, leading to a maximum of $\log_2 n + 1$ iterations.

```
function BinarySearch(list[], value)
    Integer left := 1
    Integer right := length(list)
    while left <= right
```

```
        Integer mid := floor((left+right)/2)
        if value > list[mid]
            left := mid+1
        else if value < list[mid]
            right := mid-1
        else
            return mid
        end if
    end while
    return 0
end function
```

Writing a computerized tool that calculates a correct WCEP for this program is difficult. The loop conditional compares `left` with `right`. Both these two variables are assigned to a function of `mid`, which again is assigned from a function of `left` and `right`. The particular expressions used depend on the value of `list[]`.

For a given list and value one can simply run the program, but this does not yield the WCEP, only one possible execution path. Testing all possible combinations of `list[]` is not an option either, as the number of combinations will explode even for very short lists.

In cases like this it is possible to annotate the source code with certain hints, such as the maximum number of iterations of a loop. This information can then be used to determining the WCEP.

2.2.2 The Halting Problem

While annotating source code can overcome some difficulties in determining the WCEP, the method has two problems. First, it is inelegant, and requires to explicitly specify information that is — at least in theory — possible to extract from the source code itself. Second, there is no general way to write the annotations; otherwise the annotations would not have been needed to begin with. So the problem remains: How to determine the WCEP of a general program?

In fact, it has been proved to be impossible to create a general method that can determine the execution path of any program. This is due to an important result in information theory known as the *halting problem*, originally due to Alan Turing. The halting problem originally states that it is not even possible to create a general algorithm that determines whether another program halts (ends after a finite number of operations). This result also indirectly proves that finding the exact WCET of a program is not a generally solvable problem.

The consequences of this, is that no general method can ever be found that provides the exact WCET of any program. However, as WCET analysis is mostly used in real-time systems, one may restrict the analysis to programs that may be used in real-time systems. Per definition a real-time task must finish within a given time frame, and thus must be known to halt. Knowing that a program halts does not make it any easier to show it, as shown in the above example of the binary search. But if there is a program that cannot be computationally shown to halt, that still is known to halt, then some information is known that is not apparent from the program code, and that may be used in the WCET analysis.

2.3 Execution Time Analysis

For a given execution path, what is the execution time? This problem is of a different nature than the problem of determining the execution path itself. For WCET analysis though, the problems are not independent: Finding the WCEP requires knowledge of the execution times of the different paths, otherwise one cannot know which execution path is "worst".

Measurement-based WCET analysis attempts to find an upper bound to a program's execution time by measuring execution time of one or more actual runs of the program. The alternative is to compute the execution time, or estimate it using some form of simulation. Using a cycle-correct simulator is a common approach.

2.3.1 Pipelining

The main reason that simulators need to be cycle-correct is because of pipelining effects. In a processor pipeline, the execution time of a program is less or equal to the sum of the execution times of the instructions. This is because the instructions overlap. The effect is more dramatic when there are parallel execution units; in this case doing an operation twice in a row may not take much more time than doing it once.

This makes it difficult to divide a program into parts, finding the execution time of each part separately and summing the results. At the extreme, in a processor with no pipeline one can find an accurate execution time by summing the execution times of all instructions executed. In a pipeline, this procedure would be grossly inaccurate.

Take memory access instructions, for example. Memory access stages are usually parallel to other pipeline stages, so for instance a single, but slow memory load (like a cache miss) will execute in parallel to other instructions. If the following instructions depend on the value to be loaded, they will have to wait, but if the CPU allows *out-of-order execution* (most modern processors do) then independent instructions may "overtake" the stalled load instruction, reducing the load penalty. Clever compilers will therefore always try to load a value from memory earlier than strictly necessary, so that load penalties can be masked by other instructions.

2.3.2 Cache

Cache introduces large variations in the execution time of a program. Given the same program with the same arguments and the same initial machine state, then timing effects due to cache will be the same. The variations show up because the effect of different initial machine states and arguments is unpredictable.

For data caches, the impact depends on the memory references in the program. Memory references are dependent on the alignment of data, which may change even though the program execution path stays the same.

The instruction cache adds a variation to the time it takes to load an instruction from memory. The sequence of instructions loaded is entirely determined by the program's execution path. The address of each instruction is known after compile, and is not dependent on data alignment, as is the case for data cache. This makes the number of instruction cache misses easier to predict.

The actual time it takes to load a block from memory to cache is variable. Another is that replacing a dirty block (a block that has been modified in cache only) may require that block to be written back to memory first, whereas replacing a clean block does not.

Cache and Scheduling

Another impact on cache comes from pre-emptive scheduling. When altering between executions of various tasks, scheduling will likely cause much of the cached memory used by a task to be ejected. The worst-case outcome is that every task will have its entire cache contents removed at every task switch. Each switch therefore carries the possible execution time penalty of reloading the entire cache.

For some hard real-time tasks, this is not acceptable. One work-around is to lock the cache used by an important task, so that it is guaranteed to stay there, even after task switches. Most computer architectures support some form of cache locking. Locking blocks in the cache may stabilize the execution time of an important task, but will have a negative impact on other tasks, as less cache memory is available to them.

2.3.3 Branch Prediction

On conditional branches, the next instruction cannot be known until the branch is resolved. This means that such instructions cannot automatically be pipelined, and this would lead to large performance penalties in tight loops for pipelined processors. The workaround is to use branch prediction, which guesses which path a branch will take. Branch prediction can be *static*, or *dynamic*. Static branch prediction uses different branch instructions for branches that are to be predicted taken and branches that are not. Dynamic branch prediction uses branch history to predict.

Branch prediction will sometimes predict incorrectly when guessing the next instruction. This yields a small execution time penalty. For static branch prediction, which branches that are correctly predicted, and which are not, can be found by locally examining the execution path around the branch, because in a static branch prediction scheme, predictions are per definition not dependent on branch history or other parts of the machine state. Dynamic branch prediction is much harder to analyze, because they may depend on the execution history of other, perhaps even unrelated, branches.

The exact time penalty of a single branch prediction miss is variable, and depends on the state of the pipeline at the time the branch predictor's guess was found to be wrong. The time penalty is usually nicely bounded.

2.4 Unexpected Timing Effects

This section explains some unexpected timing effects caused my modern hardware.

2.4.1 Timing Anomalies

A timing anomaly is when an effect that slows execution of part of a program turns out to speed up execution of the program as a whole, or vice versa. The first timing anomaly to be analyzed was the *scheduling anomaly*, analyzed by Graham et al. [21], in which the total time needed to schedule a set of tasks would increase, if one of the tasks had its execution time reduced. It can only happen when the scheduler algorithm is *greedy*, that is, it bases its decision on what looks best at the moment, without considering future tasks.

The scheduling anomaly is important in WCET analysis because it directly translates into the problem of scheduling out-of-order execution of instructions through a pipeline.

An example of a timing anomaly is given in Lundqvist [22]. In this example, a cache miss would result in shorter execution time than a cache hit.

The following 5 instructions are to be executed (in pseudo-assembly code, `A-E` are registers):

```
1: A := LOAD SOMETHING FROM MEMORY
2: B := A+A
3: D := C+C
4: E := D*D
5: F := E*E
```

The program runs on a computer with a separate load-store unit, a separate adder and a separate multiplier. It may load, add and multiply at the same time, but only do one of each at a time.

If the load in instruction 1 hits, then instruction 2 will be executed right away. After 2 finishes, execution will continue on instruction 3. After 3 finishes, instruction 4 will start. When instruction 4 finishes, instruction 5 will start.

If, however, the load instruction misses, then instruction 2 will stall. Now, instruction 3 will be executed instead, as it is not dependent on the result of the load. After 3 finishes, 4 and 5 will be executed sequentially.

When the load completes execution of 2 may continue. This will happen in parallel to 4 and 5, because they are independent and occupy different execution units (the summer and the adder). Because of this parallel addition, the cache miss takes less time than the cache hit. (Example is from Lundqvist [22], but edited.)

However, a decent compiler would try to reorder instructions so that a load occurs several instructions before its value is to be used. Here that can be done by delaying instruction 2, because the result of instruction 2 is not used. If instruction 2 is delayed, then the timing anomaly will not occur.

2.4.2 Unbounded Timing Effects

Unbounded timing effects are often called domino effects. A timing effect is unbounded when the time penalty added by the effect steadily increases as the program executes, for instance, by adding a penalty to all future iterations of a loop.

FIFO Caches can lead to unbounded timing effects in loops. Different initial cache states when entering a loop will not converge, possibly leading to different execution times for all iterations.

The canonical example is this: Take a 2-way associative FIFO cache and a loop that uses memory blocks `A`, `B` and `C`, in that order. All memory blocks map to the same cache set. When entering the loop, the cache set contains the elements `A` and `C`. It can then be shown that which of these elements that was loaded first determine whether there will be a miss on every access, or on every second access, for all future iterations of the loop.

Three iterations are listed in Table 2.1. Note how the different initial cache states have different cycles of cache contents. For an LRU cache the ache states will necessarily converge for the same sequence of memory references, because the most recently used elements will be the same independently of the initial state of the cache.

Memory access	Cache State[a]	Comment
	`C A`	Initial cache state
`LOAD A`	`C A`	
`LOAD B`	`A B`	Miss on B
`LOAD C`	`B C`	Miss on C
`LOAD A`	`C A`	Miss on A
`LOAD B`	`A B`	Miss on B
`LOAD C`	`B C`	Miss on C
`LOAD A`	`C A`	Miss on A
`LOAD B`	`A B`	Miss on B
`LOAD C`	`B C`	Miss on C

Memory access	Cache State	Comment
	`A C`	Initial cache state
`LOAD A`	`A C`	
`LOAD B`	`C B`	Miss on B
`LOAD C`	`C B`	
`LOAD A`	`B A`	Miss on A
`LOAD B`	`B A`	
`LOAD C`	`A C`	Miss on C
`LOAD A`	`A C`	
`LOAD B`	`C B`	Miss on B
`LOAD C`	`C B`	

[a]Left value is the earliest loaded, and the first to be replaced

Table 2.1: Unbounded timing effect when using FIFO caches. Top: `C` is older than `A`: Miss on every access. Bottom: `C` is newer than `A`: Miss on every second access.

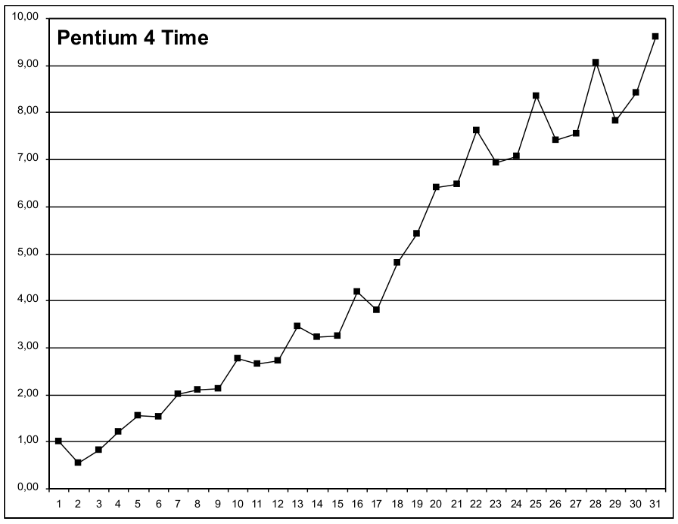

Figure 2.1: P4 Inversions in the Engblom branch prediction test. From [23]

2.4.3 Inversions and Branch Prediction

Inversions are situations where doing more work takes less time. Inversions are usually not considered a timing anomaly, as they concern scalability, and how execution times change as execution paths change; not change in execution time for a given execution path.

Interesting examples of inversions are given in Engblom [23]: A test program has a double nested loop, with the innermost loop containing nothing but a simple `nop` ("No-operation") instruction. The outer loop was fixed at 10 million iterations. The execution time of this double loop was measured for various numbers of iterations of the inner loop. This was done for several processors.

On a processor with no branch prediction, the execution time is linear with respect to the total number of cycles executed. A notable result of the experiment was that the more complex branch prediction schemes gave more erratic execution times. Pentium 4, with its intricate trace cache, had the most erratic and least monotonic graph. The execution time as a function of inner loop size is given in Figure 2.1.

There are 8 inversions in the graph: Points where an increase in the number of loop iterations reduces the execution time. The situation is extreme in this case, because there is no actual work done in the loop. It nevertheless illustrates the difficulty of making general statements about timing on complex hardware.

2.4.4 Inversions and Cache

Cache is a major cause of inversions. Measurements on a PowerPC show that a straight-forward matrix multiplication — a particularly cache-unfriendly algorithm — takes more than 3 times as long time multiplying 128×128 matrices than multiplying 129×129 matrices. There is also a smaller inversion from 64×64 to 65×65. The cause is well explained in [19].

The execution time of cached programs may also show large variations in execution times depending on the alignment in memory of variables. By only changing alignment in memory of the two factors, the highest measured execution time of a 64×64 matrix multiply was 30% higher than the lowest.

A looping program was created that could have cache miss on all accesses, or cache hits on all accesses depending only on data alignment. The source code is listed in table 2.2. This is the greatest possible alignment induced variations possible. On the target PowerPC, the highest measured execution time of the program was 24 times higher than the lowest for a high number of iterations.

The reason for these variations has to do with the way cache works: The cache on target was a 16kiB 4-way set associative LRU cache with 32 bytes line size. Each line (32 bytes) in main memory maps to only 4 lines in the cache, called *ways*. The 4 ways make up a *set*. This means that there are only four different slots in the cache which may hold any given line from memory. Because main memory is much larger than cache, many lines in main memory maps to the same set. When a line is loaded into the cache, it is loaded into the way which is least recently used (LRU).

All the elements in `array_a` that are accessed are chosen so that they map to the same set in the cache. Because there are four of them they will occupy one way each, and there will be no conflict between them. Unless they are replaced by elements from `array_b`, they will remain in the cache between iterations. The same applies vice versa to `array_b`.

The variations come from varying the alignment of `array_a` in relation to that of `array_b`: If they are aligned so that they map to different sets there will be no conflict, and no cache misses except the 8 initial. If they are aligned so that they map to the same set, they will replace each other repeatedly, and cause cache misses on every memory access.

2.5 Summary: What We Do Know

It is hard to predict the execution time of a program, or part of a program without running it, and there is no guarantee that the execution time will be the same from one run to the next.

However, if a measurement has been done on a given program, then some things will be known. This section sums up what causes variations, and what does not, if the same program is run with the same arguments.

```
for (int i = 0; i < times; ++i) {
    array_a[0]++;
    array_a[1024]++;
    array_a[2048]++;
    array_a[3072]++;
    array_b[0]++;
    array_b[1024]++;
    array_b[2048]++;
    array_b[3072]++;
}
```

Table 2.2: Code Example for Extreme Alignment Dependency

Note that not all arguments matter: For an integer matrix multiplication, for instance, only the size of the matrix and the memory alignment of the sources and results matter — not the matrix values themselves.

2.5.1 Pipelines and Branch Prediction

Complex processor pipelines and branch prediction schemes are also hard to predict. However, they cause only minor variations from one run of a program to another, as long as the execution paths remain the same.

Some heavy instructions, like floating point calculations, take a variable number of cycles to execute in the processor, depending on the numbers involved. This may cause some noticeable variations if some runs are done with "simple" numbers (for instance zeros or numbers with few non-zero bits) and some are done with entirely random numbers.

2.5.2 Cache

The impact of cache, is value-independent, but is instead highly dependent on memory alignment, as is discussed in section 2.4.4. This is most critical if a program fetches variables in several independent memory areas, and that the relative offset between those areas may change from one run to the next. Local variables are always relative to the stack pointer, which may also change between runs. It is the intensive alternation between such memory areas that cause the variations. Heavily used variables, such as counters, are often held in registers, and is not affected by cache.

If the program only uses one memory area, so that the relative offsets between variables do not change, and as long as the variables are used in the same order, then the impact of cache will not change from one run to the next. This only applies to LRU caches. For FIFO caches, the initial cache state when entering the program may cause unbounded timing penalties.

2.5.3 Conclusion

The above remarks opens up one way to estimate the execution time of a program or a function of some size (to be able to ignore pipelining effects): First run it once and measure the execution time. Also store all relevant program arguments. The next time, if the relevant arguments have not changed, the execution time will be the same.

Part II

Langauge Design

Chapter 3

TIME/occam

The TIME/occam language builds on occam while adding a `TIME` statement. It was natural then, that the language would be similar to occam. The language was also made to be simple to parse and to compile to C++ code. The TIME/occam language is a proof-of-concept language, and has fewer features and more restrictions than occam.

This chapter contains a complete description of the TIME/occam language, and explains some of the design choices. The chapter begins with an introduction to TIME/occam followed by a brief introduction to formal languages.

During the description of the language, statements and operators will often be compared with C or occam, and not explained in detail beyond that. Refer to a C reference (for instance K&R [24]) and the occam reference (*occam 2.1 Reference Manual* [25]) for more detailed descriptions.

Part of this chapter is automatically generated by `bnfc`. This applies to most of the BNF-descriptions, lexical regular expressions and lists of keywords and operators.

For the complete BNF of TIME/occam, see appendix A.

3.1 Introduction to TIME/occam

The central idea in TIME/occam is the `TIME` statement. The statement takes a time duration and a block of code. It tells the compiler/scheduler that the block of code is to be executed in within the specified time. If execution of the block communicates with other processes through channels, then those processes must also be executed within the specified time.

The statement also means that code following the `TIME` block shall not be executed before the time has elapsed, making it possible to create periodic tasks by putting the task in a `TIME` block in a `WHILE` block. The `TIME` statement will be explained in detail in chapter 4.

The basic principle of the TIME/occam is this:

Code that has no timing requirement will not be executed.

The idea behind this is that all tasks in a program have some timing constraints, and that most also have some slack. There is always a little time from a user presses a button until she expects, or can even notice, a response. The programmer is required to be conscious of timing for all parts of a system, and not only the parts in which the requirements are obvious.

With these principles, it is possible to specify the real-time requirements of a system completely within the programming language. The language, and this new way of formulating timing requirements is intended to be an experiment.

3.2 Formal View of Languages

A language is, from a formal point of view, a set of token and syntax rules, together with semantics. In natural (human) languages, the tokens are words, and the syntax rules are the way in which words may be combined into sentences. Rules associated with tokens are called lexical, and rules associated with syntax are called grammatical. Semantics concern the meaning of sentences.

For natural languages, lexical rules are the legal spelling of words, and which words are of which types (verb, adjective etc.). Grammatical rules specify the various legal ways in which types of words may be combined.

For instance, the sentence "Cat very the big is" has a grammatical error, namely an invalid ordering of tokens. The grammatical composition "*Noun adverb article adjective singular-verb*" is not a valid sentence. One of the grammatical rules that are broken in this sentence is that an article is always followed by a noun, possibly with adjectives in between: The 'the' must be followed by 'cat', possibly with an adjective (such as 'big') in between. All other uses of 'the' are invalid.

The sentence "The cat bbllllrrrrg very zzzzp." contains invalid tokens, and therefore has lexical errors. This makes it meaningless to even discuss whether or not it is grammatically correct (is "bbllllrrrrg" a verb?).

Using the grammar (the set of all grammatical rules) one may *parse* statements, that is, deduce their meanings. Parsing the sentence "The cat is very big" we learn that some thing "is". What "is"? "The cat" is. What "is the cat"? The cat is "very big".

3.2.1 Ambiguity

Sentences in natural languages can be ambiguous in many ways. The sentence "I shot an elephant in my pyjamas" is ambiguous to whether "I" or the elephant was wearing the pyjamas. When hearing such a sentence, a person would know that the description "in my pyjamas" does not fit the context of “elephant”, even though this is not expressed directly. The ambiguity in the sentence arises from unclear grouping of words; it can either be “(I shot an elephant) (in my pyjamas)” or “I shot an (elephant in my pyjamas)”.

Some ambiguities are so subtle that they hardly change the meaning of a sentence at all. “My friends left early last night” has two possible groupings: “(My friends) left (early last night)” or “(My friends) (left early) (last night)”.

Other ambiguities completely change the meaning of a sentence. “I want apples and pears or bananas” has the possible groupings “I want apples and (pears or bananas)”, or “I want (apples and pears) or bananas”. In the first you want apples, and in addition to the apples, either pears or bananas. In the second you want apples and pears, but if you cannot get them both, you want bananas.

3.2.2 Mathematical Notation

Mathematical notation is a language. The tokens are numbers, identifiers and operators. Unlike natural languages there is no room for ambiguity. A statement in mathematical notation must be uniquely defined. To fix grouping ambiguities, special rules apply for operators.

For instance, $2 \times 3 + 4$ groups as $(2 \times 3) + 4$, because the multiplication operator is defined to have a higher *precedence* than addition. Expressions with higher precedence are grouped before expressions with lower precedence. This clears many ambiguities in cases that involve different operators.

There is, however, still the question of how to group the several expressions of the same operator, or with operators with the same precedence. Examples are $2 - 3 - 4 - 5$ and 2^{3^4}. The rules regarding this are called *associativity rules.* Subtraction is defined to be left associative, and parses as $((2 - 3) - 4) - 5$, while the exponentiation is right associative and parses as $2^{(3^4)}$. Note that multiplication and addition is freely associative, and that the grouping is irrelevant to the mathematical meaning of the expression: $2 + (3 + 4)$ has the same value as $(2 + 3) + 4$. In mathematics, this attribute is confusingly called “associative”, like in “addition is associative”.

In mathematics, parenthesis is part of the language, and can be used to override the default rules. If we really want $2 - (3 - 4)$, we write just that.

3.2.3 Backus-Naur Form

Backus-Naur form is a notation for describing grammar. It was first used to describe the then new programming language ALGOL in 1958, and has since become the de facto way to describe formal languages.

The syntax is quite simple, and can best be explained by examples. For instance, expressions describing addition and subtraction of numbers can be described by the following BNF:

$$\begin{array}{lcl} \langle \mathit{Exp} \rangle & ::= & \langle \mathit{Exp} \rangle + \langle \mathit{Num} \rangle \\ & | & \langle \mathit{Exp} \rangle - \langle \mathit{Num} \rangle \\ & | & \langle \mathit{Num} \rangle \end{array}$$

The first line reads: An ⟨*Exp*⟩ can be constructed from another ⟨*Exp*⟩ followed by the character "`+`" followed by a ⟨*Num*⟩. The "|" means "or", so the second line is a similar way to construct an ⟨*Exp*⟩; from an ⟨*Exp*⟩, a "−" and a ⟨*Num*⟩. The last line says that an ⟨*Exp*⟩ may be constructed from a single number, a ⟨*Num*⟩. Each way to construct an ⟨*Exp*⟩ is called a *production rule*.

Note the recursive definition, allowing more than one addition/subtraction in the expression. The symbols ⟨*Exp*⟩ and ⟨*Num*⟩ are called *non-terminals*, because there exist rules to substitute them with something else. "+" and "−" are *terminal* symbols, because no such rules exist. Before parsing an expression, it must be "lexed" into a list of terminal symbols.

Grammar definitions span what are called *abstract syntax trees*, or ASTs. The abstract syntax tree of the above grammar is depicted in figure 3.1. The AST is useful as basis for a data structure to hold parsed grammar.

After successfully parsing a specific sentence, one obtains the *concrete syntax tree*.

Take the text "$1 + 3 - 2 - 4$". Lexing it finds the sequence "`Num(1) "+" Num(3) "−" Num(2) "−" Num(4)`". Parsing it with the above grammar and one gets the concrete syntax tree depicted in figure 3.2.

3.2.4 Associativity and Precedence in BNF

In the example BNF in the previous section, a subtraction is defined as:

$$\langle \mathit{Exp} \rangle \quad ::= \quad \langle \mathit{Exp} \rangle - \langle \mathit{Num} \rangle$$

and not

$$\langle \mathit{Exp} \rangle \quad ::= \quad \langle \mathit{Exp} \rangle - \langle \mathit{Exp} \rangle \qquad\qquad \longleftarrow \text{Wrong!}$$

This is because the latter does not enforce left associativity as the first do, which means that an expression like "$1 - 2 - 3$" in the latter grammar may be incorrectly parsed as "$1 - (2 - 3)$". The first grammar fixes this because the right hand side of the subtraction is a ⟨*Num*⟩ and not an ⟨*Exp*⟩, so

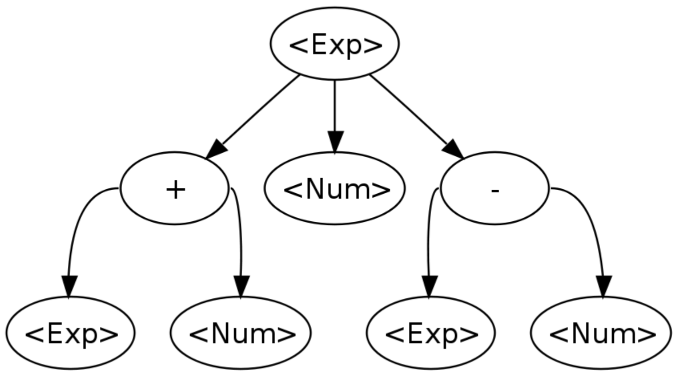

Figure 3.1: Example of abstract syntax tree

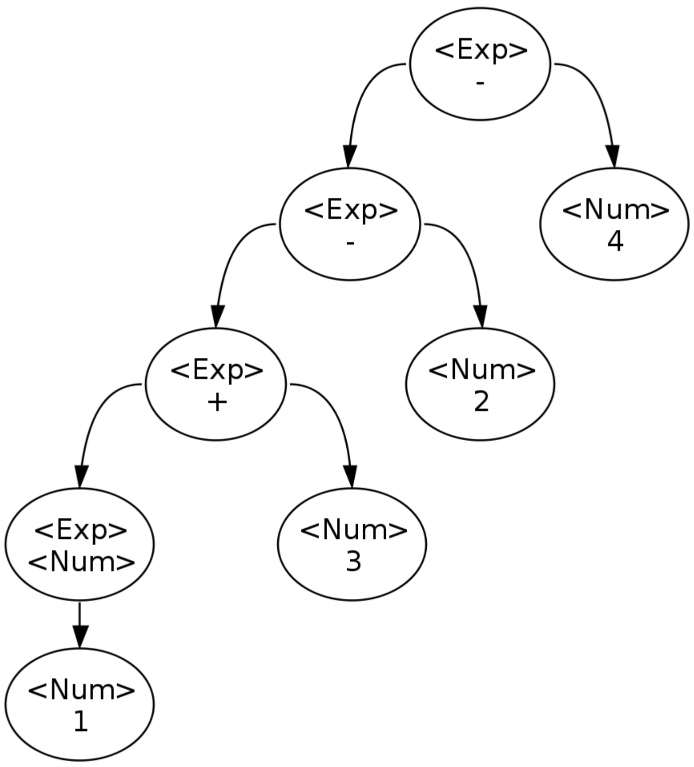

Figure 3.2: Example of concrete syntax tree, for $1 + 3 - 2 - 4$.

the result of a subtraction ($\langle Exp \rangle$) may not be used as a right hand side of another subtraction.

This is an example of how associativity must be implicitly expressed in BNF. There is no explicit way to define precedence rules either. If we add multiplication and exponentiation to the above grammar, we cannot just write it as:

$$
\begin{array}{lcll}
\langle Exp \rangle & ::= & \langle Exp \rangle \texttt{ + } \langle Num \rangle & \\
 & | & \langle Exp \rangle \texttt{ - } \langle Num \rangle & \\
 & | & \langle Exp \rangle \texttt{ * } \langle Num \rangle & \longleftarrow \text{Wrong!} \\
 & | & \langle Exp \rangle \texttt{ ^ } \langle Num \rangle & \longleftarrow \text{Wrong!} \\
 & | & \texttt{(} \langle Exp \rangle \texttt{)} & \\
 & | & \langle Num \rangle &
\end{array}
$$

This would not specify any precedence information, so "$1 + 2 \times 3 + 4$" may be grouped as "$(1 + 2) \times (3 + 4)$", and still conform to the grammar.

To systematically express precedence and associativity in BNF, a common way is to distinguish between symbols of varying precedence using a numeric index. A constant (like $\langle Num \rangle$) has the highest index, and so does bracketed groups. Rules are added to let symbols have their precedence decreased, but not increased. A binary operator with precedence p should have result level p. If it is left associative, it should have a left hand argument at level p, and a right hand argument at level $p + 1$, and vice versa if it is right associative.

A correct way of specifying mathematical expressions with addition, subtraction, multiplication, exponentiation and parenthesis is then:

$$
\begin{array}{lcl}
\langle Exp1 \rangle & ::= & \langle Exp1 \rangle \texttt{ + } \langle Exp2 \rangle \\
 & | & \langle Exp1 \rangle \texttt{ - } \langle Exp2 \rangle \\
 & | & \langle Exp2 \rangle \\
\langle Exp2 \rangle & ::= & \langle Exp2 \rangle \texttt{ * } \langle Exp3 \rangle \\
 & | & \langle Exp3 \rangle \\
\langle Exp3 \rangle & ::= & \langle Exp4 \rangle \texttt{ ^ } \langle Exp3 \rangle \\
 & | & \langle Exp4 \rangle \\
\langle Exp4 \rangle & ::= \texttt{(} & \langle Exp1 \rangle \texttt{)} \\
 & | & \langle Num \rangle
\end{array}
$$

There are now 4 levels of $\langle Exp \rangle$. The relevant constants (here only $\langle Num \rangle$) have the highest precedence level.

The result of an addition (level 1) is now lower than any of the arguments to multiplication (level 2 and 3), so the result of an addition cannot be used in multiplication. But because the result of a multiplication (level 2) may be used to produce a level 1 (by the rule ⟨*Exp1*⟩ ::= … | ⟨*Exp2*⟩), the reverse is possible. The consequence is that in expressions like "$1 + 2 \times 3 + 4$", the multiplication must be grouped first. The same applies to exponentiation and multiplication. This resolves operator precedence.

The above grammar also fixes associativity. For example, the '$-$' operator has a result of level 1, a left hand side argument of level 1, but a right hand side argument of level 2. This means that the result of a subtraction may only be used as left hand side of another subtraction. The subtractions in a statement like "$1 - 2 - 3$" must therefore be evaluated from left to right. It is the other way with exponentiation, which is right associative, and which result may only be used as a right hand side of another exponentiation.

Anything in parenthesis has the highest precedence, and may be left or right operant to any operators.

3.2.5 Lists and Repetition in BNF

A grammar often includes repeated elements; it may be that a program consists of several statements, or that a sentence includes several words. In BNF, lists are defined recursively. A sentence, that may be one or more words followed by a period, may be defined as follows:

```
⟨Sentence⟩       ::=  ⟨List-of-words⟩ .
⟨List-of-words⟩  ::=  ⟨Word⟩
                  |   ⟨Word⟩ ⟨List-of-words⟩
```

In some lists, items are separated by a symbol, for instance ",". The symbol is called a *terminator* if it follows all elements of the list, including the last; or a *separator* if it follows all elements except the last. The sentence definition above has neither separator nor terminator.

In some cases, a non-empty list is accepted, and in other cases it is not. The sentence definition above has a non-empty list definition. Various kinds of lists may be defined as follows (examples from definition of LBNF [26]):

$$
\begin{array}{lll}
\langle \mathit{Separated} \rangle & ::= & \epsilon \\
 & | & \langle \mathit{Item} \rangle \\
 & | & \langle \mathit{Item} \rangle \ \texttt{,} \ \langle \mathit{Separated} \rangle \\
 & & \\
\langle \mathit{Nonempty\text{-}separated} \rangle & ::= & \langle \mathit{Item} \rangle \\
 & | & \langle \mathit{Item} \rangle \ \texttt{,} \ \langle \mathit{Nonempty\text{-}separated} \rangle \\
 & & \\
\langle \mathit{Terminated} \rangle & ::= & \epsilon \\
 & | & \langle \mathit{Item} \rangle \ \texttt{;} \ \langle \mathit{Terminated} \rangle \\
 & & \\
\langle \mathit{Nonempty\text{-}terminated} \rangle & ::= & \langle \mathit{Item} \rangle \ \texttt{;} \\
 & | & \langle \mathit{Item} \rangle \ \texttt{;}
\end{array}
$$

Note that an empty symbol in BNF is the "ϵ".

3.3 TIME/occam Lexical Definitions

This section explains the basic lexical rules of the language TIME/occam, including layout syntax, case sensitivity, reserved keywords, operators and token definitions.

3.3.1 Layout Syntax

Occam has a rigid layout system, which requires exactly two spaces of extra indentation for every nested block. The compiler tool used to create the TIME/occam compiler, `bnfc`, does not support such a system. To keep the implementation of the layout system simple, this feature of occam had to be changed.

Instead, TIME/occam has a layout system similar to Haskell and Python. Every statement that is followed by a block (like `IF` or `PROC`) is first followed by a colon (`:`), like in Python. Otherwise, the layout system is like in Haskell:

- Statements and blocks can be specified using semicolons, opening and closing braces, like in C. Layout only affects blocks with no explicit braces and semicolons.

- Every block has a base indentation. When a colon is encountered, unless the next token is "`{`", a "`{`" is added and the next token specifies base indentation of a nested block. The base indentation of a nested block must be larger than the base indentation of the parent block.

- Before every token that begins on the base indentation, a semicolon is added.

- If a token begins on an indentation less than the base, "`}`"s are inserted before it, until matching the corresponding "`{`" at the same indentation.
- To let a statement continue on the next line, all one needs to do is to have the next line have more indentation than block base.

The layout is processed by a layout resolver, which adds curly braces and semicolons to the text. After resolving layout, which happens between lexing and parsing, blocks are started by "`{`", and ended by "`}`"; and statements are separated (not terminated) by "`;`". The layout resolver code is part of `bnfc`.

An example of before and after the layout resolver is given in figure 3.3.

3.3.2 Identifiers and Case Sensitivity

TIME/occam has strict case rules. All types and keywords must be uppercase (except `true` and `false`, because they could just as well be variables). Upper case identifiers have the BNF name ⟨*UpperIdent*⟩.

All procedures (defined by `PROC`) must have capitalized names; an initial capital letter followed by at least one lower-case letters. These are called ⟨*CapIdent*⟩ in the BNF description. All variables and functions must begin with a lowercase letter, and are called ⟨*LowerIdent*⟩.

The reason for the strict case rules is that it makes it simpler to create unambigous grammar. Knowing that types are all-uppercase, means that one can distinguish between declarations like "`INT [10] var`" and assignments like "`var [10] = 0`" without needing multiple token lookahead[1]. Now, any statement beginning with an all-uppercase identifier is a declaration.

Similarly with the capitalized parametric processes; the difference between calling a parametric process "`MyProc(var1, var2)`" and calling a function "`func(var1, var2)`" can be seen immediately. The compiler can sort out the difference without the need for further parsing.

Underscores are allowed in identifiers, but not as first letter. In contrast to occam, periods are not allowed as part of identifiers, although there is no particular reason why they may not be.

The regular expression patterns for ⟨*CapIdent*⟩, ⟨*LowerIdent*⟩ and ⟨*UpperIdent*⟩ can be written as follows: (upper, lower, letter, digit are single upper-case, lower-case, or any case letters, or a single digit, respectively.)

CapIdent: ⟨*upper*⟩(⟨*upper*⟩ | ⟨*digit*⟩ | '_') ∗ ⟨*lower*⟩(⟨*letter*⟩ | ⟨*digit*⟩ | '_')∗

LowerIdent: ⟨*lower*⟩(⟨*letter*⟩ | ⟨*digit*⟩ | '_')∗

UpperIdent: ⟨*upper*⟩(⟨*upper*⟩ | ⟨*digit*⟩ | '_')∗

[1]Lookahead is explained in section 7.1.1.

```
PROC TestIf():
    IF b == 0:
        PRINT "This is one very long statement ",
            "printed over several lines"
        PRINT "Statements"; PRINT " on "; PRINT "the same line"
    ELSE:
        IF c == 0:
            IF d == 0: {Do_something()}; ELSE: {Do_something_else()}
        ELSE IF c == 1: {
                    Do_something_completely_different()
         ;
        Do_yet_more()
                                                 }
```

```
PROC TestIf () : {
  IF a == 0 : {
    IF b == 0 : {
      PRINT "This is one very long statement ", "printed over several lines" ;
      PRINT "Several statements" ;
      PRINT " may" ;
      PRINT "be on the same line"
    } ;
    ELSE : {
      IF c == 0 : {
        IF d == 0 : {
          Do_something ()
        } ;
        ELSE : {
          Do_something_else ()
        }
        } ;
      ELSE IF c == 1 : {
        Do_something_completely_different ()
      }
      } ;
    Do_yet_more ()
  }
  }
```

Figure 3.3: Example of resolving layout.
Above: Before the layout resolver. Below: After the layout resolver.

3.3.3 Literals

- String literals ⟨*String*⟩ have the form `"`x`"`, where x is any sequence of any characters except `"` unless preceded by `\`.

- Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

- Double-precision float literals ⟨*Double*⟩ have the structure indicated by the regular expression ⟨*digit*⟩+ '.'⟨*digit*⟩+('*e*''-'?⟨*digit*⟩+)? i.e. two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

- Character literals ⟨*Char*⟩ have the form `'`c`'`, where c is any single character.

3.3.4 Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in TIME/occam are the following:

`ALT`	`BOOL`	`CHAN`
`CINCLUDE`	`CONST`	`CSYSINCLUDE`
`DELAY`	`ELSE`	`FOR`
`IF`	`IN`	`INT`
`OUT`	`PAR`	`PRINT`
`PROC`	`REAL`	`SEQ`
`SKIP`	`STOP`	`STRING`
`TIME`	`TO`	`TYPE`
`VAL`	`WHILE`	`false`
`true`		

The symbols used in TIME/occam are the following:

```
=      [      ]
;      ,      (
)      :      {
}      !      ?
&      *=     /=
%=     +=     -=
<<=    >>=    &=
^=     |=     ||
&&     |      ^
==     !=     <
>      <=     >=
<<     >>     +
-      *      /
%
```

3.3.5 Comments

- Single-line comments begin with −−.
- Multiple-line comments are enclosed with `/*` and `*/`.

3.4 Blocks, Declarations and Top-level Elements

This section describes the language elements of TIME/occam that are not processes or expressions.

3.4.1 Blocks

There are two types of blocks in TIME/occam. One is called a single-statement block. It has no ordering (SEQ/PAR), and is only allowed to contain one statement. The other type of block must be a direct sub-block of either a SEQ or PAR statement. It may contain a list of statements.

Both types of blocks may begin with a list of declarations. Declarations are only allowed at the beginning of a block, before any non-declarative statements.

3.4.2 Declarations

Declarations are found in parameter lists for procedures, and as variables at the beginning of blocks. A declaration consists of a type specification, an optional array specification, an optional qualifier and a name.

- Valid types are `STRING`, `BOOL`, `INT`, `REAL`, and custom types. Custom types are defined by the `TYPE` keyword (see section 3.4.3).

- Only one-dimensional arrays are supported. Arrays with empty brackets are allowed in parameter lists, and means any array is accepted as argument.
- Variables may be initialized. Arrays may only be initialized to a single value (which means that all elements of the array will be initialized to that value).

The qualifiers have the following meaning:

`CONST`	The variable cannot be changed. For parameters it also implies that the argument is taken by reference.
`CHAN`	The variable is a channel. The type of a channel is the type of values that can be sent through the channel. All channel parameters are taken by reference. Using this qualifier on a new variable creates a new channel.
`IN`	Only valid for parameters: The variable is a channel, restricted to receive.
`OUT`	Only valid for parameters: The variable is a channel, restricted to send.
`VAL`	Only valid for parameters: The parameter is taken by value.
None	This denotes a changeable variable or a changeable parameter taken by reference.

3.4.3 Top-level Elements (PROC, TYPE, CINCLUDE)

A program consists of a list of top-level elements, which may be procedures, type definitions, or inclusion of C-headers. Execution of the program starts at the procedure named `Main`.

The order of top-level elements is not important. Types and procedures need not be defined before they are used.

Procedures

Procedures are callable processes with parameters. They do not return a value, and are defined with the `PROC` keyword.

Example:

```
PROC Main(STRING [] args):
    PRINT "Hello World!"
```

Type definitions

Type definitions are defined using the `TYPE` keyword, and are similar to `typedef`s in C. Types may be defined in terms of another type, and an optional array specifier.

Examples:

```
TYPE DOUBLE = REAL
TYPE BUFFER = INT [10]
```

The support for arrays is at the moment limited to a single dimension, which will cause troubles if creating an array of a new type, which is itself an array, as only one dimension may be referenced at a time. A workaround is to use temporary variables.

Including C Headers

C headers can be included with the `CINCLUDE` or `CSYSINCLUDE` command. The first command searches in the current directory for headers, the latter searches in system directories.

Examples:

```
CINCLUDE "my_header.h"
CSYSINCLUDE "unistd.h"
```

3.5 Expressions

Expressions in TIME/occam are basically C-expressions, without any operators that modify operands. Assignments are statements, and are not allowed as expressions. The prefix and postfix increment/decrement operators have been removed. This simplifies program analysis somewhat, limiting the places in a program where variables may change.

There is another important difference between TIME/occam expressions and C expressions: The operator precedence of the relational operators has been decreased, to allow bit masking tests without parenthesis:

In C, the expression "`a & 0xFF == 0`" will never become true, because the equals-to operator has a higher precedence than the bitwise-and. Such bit masking tests are common in embedded programming; and in C, they require an extra parenthesis around the left hand side of the equivalence operator. By reducing the precedence of all the relational operators (`== != <= >= < >`) these extra parenthesis can be avoided.

A list of operators and precedence is given in table 3.1. The BNF is not listed, but can be found in appendix A.

3.6 Processes and Statements

The CSP term for what in other programming languages is called a statement is "process". There are primitive and constructed processes, the latter are built up by control statements and primitive processes.

The primitive processes are assignment, send, receive, `SKIP`, `STOP`, standalone expressions, and the utility processes `DELAY` and `PRINT`. The constructed processes are variants of `SEQ`, `IF`, `WHILE`, `PAR`, `ALT` and `TIME`.

Operator	**Symbol**	**Precedence**
Logical OR	`||`	1 (Lowest)
Logical AND	`&&`	2
Equal to	`==`	3
Not equal to	`!=`	3
Bitwise OR	`|`	4
Bitwise XOR	`^`	5
Bitwise AND	`&`	6
Less than	`<`	7
Greater than	`>`	7
Less than or equal to	`<=`	7
Greater than or equal to	`>=`	7
Bitwise left shift	`<<`	8
Bitwise right shift	`>>`	8
Add	`+`	9
Subtract	`-`	9
Multiply	`*`	10
Divide	`/`	10
Modulus	`%`	10
Negate	`-`	11
Positive	`+`	11
Index	`[]`	12
Function call	`()`	12

Table 3.1: List of TIME/occam Expression Operators

Occam does not have `TIME`. TIME/occam does not have the occam process `CASE`.

3.6.1 Assignment

Assignments work like in C, and not like in occam. Multiple assignments are not allowed in one statement, and the basic assignment operator is "`=`" and not "`:=`".

Compound assignment is allowed, and uses the left hand side of the assignment as left side of the operator, so that "`a += b`" is equivalent to "`a = a + b`". The left hand side may be a variable or an array element.

Examples:

```
a = 10
array[a] = b+c
```

3.6.2 Communication

Send and receive works like in occam, with some major restrictions.

- You *cannot* send or receive from an element of an array of channels. Elements of channel arrays may only be passed to procedures. This restriction is explained in section 9.2.1.
- You cannot send an expression, only the value of a variable. This is because sending is currently implemented as passing a pointer.

Unlike occam, where channels are used to form parallel algorithms, channels in TIME/occam are mainly intended to be a synchronization and communication mechanism between parallel threads. The current compiler does not utilize multiple processors.

Example of send/receive:

```
CHAN INT ch
INT x = 1
INT y
PAR:
    ch ! x
    ch ? y
```

3.6.3 SKIP and STOP

`SKIP` does nothing and terminates, while `STOP` does nothing forever. `STOP` is equivalent to

```
WHILE 1:
    SKIP
```

3.6.4 Executing Expressions

An expression is a valid process, even though its result is not used. This is useful for running external C functions that has side effects and no return value.

Example:

```
INT x = 1
WHILE 1:
    SEQ:
        x ^= 1
        write_io(x)
```

3.6.5 Utility processes, DELAY and PRINT

`DELAY` pauses execution the given time (in milliseconds). It is similar to the C library function `usleep`. `PRINT` prints a list of expressions to standard output. The expressions are printed with no implicit spacing, but a trailing newline is added.

Examples:

```
INT ctr = 0
WHILE 1:
    SEQ:
        PRINT "Hello, the counter is ", ctr
        DELAY 1000
        ctr += 1
```

3.6.6 SEQ And PAR

It is not legal, neither in occam or TIME/occam, to list several actions without specifying whether or not they are to be executed in sequence or in parallel. This is specified with the keywords `SEQ` and `PAR`.

Like in occam, `SEQ` and `PAR` may be replicated with the `FOR` keyword. Replicating a sequence yields a sequence of processes, like a standard C `for` loop. Replicating with the `PAR` keywords yields a number of processes running in parallel.

Like in occam, the use of a `FOR` is more limited than a C loop. The iterator must be an integer, and is read-only to other code. There can only be specified an initial value and an upper bound, and the iterative step is always to increment the iterator.

3.6.7 IF and ELSE

Conditionals are written using the two keywords `IF` and `ELSE`. Unlike in occam, `IF` cannot be replicated (there is no `IF FOR`).

Example:

```
IF x == 0:
    PRINT "Zero!"
ELSE IF x == 1:
    PRINT "One!"
ELSE:
    PRINT "Something else: ", x
```

3.6.8 WHILE

A `WHILE` block is a loop that will keep executing as long as the given condition is true.

Examples:

```
WHILE 1:
    TIME 10:
        PRINT "This is a periodic task"

WHILE abs(func(x)) > 10e-3:
    x = guess(x)
```

3.6.9 TIME

The `TIME` statement takes a time duration and a block of code. The statement tells the compiler/scheduler that the block of code is to be executed in within the specified time.

The statement also means that code following the `TIME` block shall not be executed before the time has elapsed. See chapter 4 for in-depth information on the consequences of the `TIME` statement.

Example:

```
WHILE 1:
    TIME 10:
        SEQ:
            REAL tmp = measure()
            REAL output = calculate_output(tmp)
            set_output(output)
```

3.6.10 ALT

`ALT` works much like in occam, choosing from a list of channels to receive from. The lists of channels may be preceded by a boolean guard and a “`->`”, disabling or enabling choice of that channel. A `SKIP` may also be guarded; if the `SKIP` is active, the `ALT` will pass any alternatives and skip to the next statement.

`ALT` may be nested and replicated; although replication is of limited use because of the limitation of channel arrays (see 9.2.1).

Example:

```
INT dummy
ALT:
    ALT FOR i = 0 TO 10:
        array[i] < 5 -> ch1 ? dummy:
            PRINT i
    a && b -> ch1 ? dummy:
        PRINT dummy
    c == 0 -> ch2 ? c:
        SKIP
    d != 0 -> SKIP
```

Chapter 4

The TIME Statement

The `TIME` statement places a time limit on completing the specified block of code, and all other code that the block depends on. It also specifies that code following the `TIME` statement in sequence should not begin executing before the specified time has elapsed.

4.1 Details of the TIME Statement

As mentioned in section 3.1, code that has no timing requirement will not be executed.

4.1.1 Discovery and Control Processes

There has to be exceptions to this rule. Some code must be run without timing requirements, or else the `TIME` statements will not be discovered. For instance, there cannot be a timing requirement on `PROC Main()`, because there cannot be a `TIME` statement preceding it. Yet, `PROC Main()` must of course be run, or else no `TIME` statements will ever be discovered.

Processes that are executed in search of `TIME` statements are called *discovery processes*. This includes the following:

- `SEQ` [`FOR`]
- `PAR` [`FOR`]
- `WHILE`
- Procedure call

The process of finding `TIME` statements that are only preceded by discovery processes is called *discovery*. Discovery stops when it encounters a process that is not a discovery process.

Example: In the below code, only “`Hello `” will be printed, because the other `PRINT` is not in a timer that will be discovered.

```
PROC Proc():
    TIME 1:
        PRINT "Hello "

PROC Main():
    INT x
    PAR:
        Proc()
        SEQ:
            x = 1
            TIME 2:
                PRINT "world!"
```

4.1.2 TIME and Channels

If a process with a timing requirement sends or receives from a channel, the code preceding the other end of the channel must also be executed, even if it is not itself in a `TIME` block.

Example: In the below code, "`Hello world!`" will be printed.

```
PROC Proc(IN INT ch):
    INT dummy
    SEQ:
        PRINT "Hello "
        ch ? dummy

PROC Main():
    CHAN INT ch
    INT dummy = 1
    PAR:
        Proc(ch)
        TIME 1:
            SEQ:
                ch ! dummy
                PRINT "world!"
```

4.2 Missed Deadlines

Explicit timing requirements in code also allows for explicit handling of missed deadlines. Missed deadlines should normally be handled as error conditions, as discussed in section 1.2.3. The syntaxes presented in this section have not been implemented or formalized. These alternatives are not mutually exclusive, and may all be part of the language. The programmer is then free to choose the most suitable way to handle deadlocks for a particular application.

4.2.1 Priorities

One option is to give priorities to `TIME` statements. These will only be used in case not all timing requirements can be guaranteed and scheduling will otherwise still be done by earliest deadline first. This use of priority is in line with the common meaning of the word: If the scheduler must prioritize between the threads, then it will run the highest priority threads first.

Sticking to EDF for scheduling means that a higher utilization is possible, than if priorities were to be used. An example of possible syntax is:

```
TIME 10 PRI 1:
    ...
```

4.2.2 Treating as Exceptions

Another option is to treat a missed deadline as an exception, in line with exception handling in languages like Java and C++; and terminate the execution of code in the TIME-statement. This could be done with a `MISS` statement:

```
TIME t:
    Timed_code()
MISS:
    Exception_code()
```

What it would mean to terminate the execution of a block must be carefully defined. As with general exception handling, it can be challenging to cut execution without leaving potential memory leaks. Not only must variables from the `TIME` block be deleted, but also variables from other blocks, connected to the `TIME` block by channels, if that code is no longer needed.

One use of the `MISS` statement is to simply abort if the deadline cannot be reached. This can be known in advance, during the planning phase. Aborting one `TIME` block means that more execution time is available to other blocks. `TaskCode()` does not execute or complete unless all timing requirements of the entire program can be guaranteed. When the timer expires, code will continue beyond the `TIME` statement, which restarts the task. `MISS:{SKIP}` is therefore a way to flag the task as lowest-priority.

```
WHILE 1:
    TIME 10:
        TaskCode()

    MISS:
        SKIP
```

4.2.3 Weakly Hard Timing Requirements

Using the `MISS` statement, one can create the timing requirement $\binom{n}{m}$, described in section 1.1.3. This timing requirement means that for any m deadlines of a periodic task in a row, at least n of them must be met. In TIME/occam this can be formulated with the following code:

```
INT met = 0
INT count = 0
WHILE:
    TIME t:
        SEQ:
            count = count + 1
            Code()
            met = met + 1
    MISS:
        IF n - met == m - count:
            SEQ FOR i = met TO n:
                TIME t:
                    Code()
```

This assumes that "`count = count + 1`" will always be run, and that "`hits = hits + 1`" will only be run if the deadline is met. It also assumes that $n < m$. The basic idea is that because the `TIME` statement in the `SEQ FOR` loop does not have a `MISS` statement, then it will never be skipped.

Similar code can be made for $\left\langle {n \atop m} \right\rangle$, $\overline{\binom{n}{m}}$ and $\overline{\left\langle {n \atop m} \right\rangle}$. Better still is to create a clear syntax for weakly hard timing requirements. The following code contains suggestions to syntax for the four weakly hard timing constraints:

```
PAR:
    WHILE 1:
        TIME t1 MEET ANY n1 IN m1:
            ...
    WHILE 1:
        TIME t2 MEET ROW n2 IN m2:
            ...
    WHILE 1:
        TIME t3 MISS ANY n3 IN m3:
            ...
    WHILE 1:
        TIME t4 MISS ROW n4 IN m4:
           ...
```

The scheduler should distribute missed deadlines "fairly" among tasks, so that not one timing requirement has a maximum allowed number of misses, while another has none. This is discussed in Bernat et al. [2].

Chapter 5

TIME/occam Scheduler

Central to running TIME/occam programs is a scheduler that can use the information from the `TIME` statements. This scheduler is not an operating system scheduler, but is compiled and linked with the program itself.

The scheduler has the following tasks:

- Build up a graph of future execution and timing requirements. This is the prediction phase.
- Serialize parallel execution in a way that guarantees, if possible, that timing requirements are met. The entire program is serialized into a single chain of processes, even if the processes are independent. This is also part of the prediction phase.
- Deal with deadlines that are missed in some specified way; or delay or slow down execution if necessary, so that processes are not executed too early. This is the planning phase.
- Execute the program using the new schedule.

5.1 Select Algorithm

When running a parallel program on a single processor, the program must be serialized.

The TIME/occam scheduler algorithm is a form of Earliest Deadline First (EDF). The difference between it, and normal EDF, is that the TIME/-occam scheduler attempts to predict execution some time into the future before beginning execution. If timing requirements can be guaranteed with slack, the scheduler may add execution pauses or slow down execution. If the timing requirements fail, the scheduler can take the appropriate action. The amount of time into the future that the scheduler tries to predict is called the *prediction horizon.*

The select algorithm controls which process that is to be serialized next. The pseudo code is as follows:

1. From `PROC Main()`, discover all `TIME` blocks that are preceded only by control processes. Control processes are described in section 4.1.1. This gives the initial list of timers.

2. Choose a prediction horizon.

3. Select the timer with the earliest deadline. If the earliest start-time of that timer is in the future, the time to the start of that timer is the new temporary prediction horizon. This is because another task that is chosen to run does not have the earliest deadline, only the earliest deadline of all tasks that are ready. When the task with the earliest deadline is ready, it must be chosen, so when it is ready, the scheduler must return to select. This is achieved by limiting the prediction horizon. An earliest start time in the future occurs if a `TIME` statement follows another `TIME` statement, as the second is not allowed to start until the first timer expires.

4. Select the timer with the earliest deadline, whose start time is not in the future. If there are none, insert a delay into the serialization long enough to make the first timer ready.

5. Select the next process to be serialized for that timer. This is the first in the `TIME` block if this is the first serialization of that timer.

6. Run the serialization algorithm on that process. The algorithm returns the next process to be serialized, if known, and a state of one of the below flags:

   CONTINUE means that a process has been serialized, and that the next process to be serialized is known. Update the "next" value of the active timer. Reduce the remaining horizon, and then repeat point 6 with the next process.

   RESELECT means that there is now a choice to which process to serialize next. This state appears when encountering forks (such as `PAR` or send/receive), or if a new timer is found. Return to point 3.

   COMPLETED means that the current timing horizon has been completed. If this was a temporary horizon because the earliest deadline timer was in the future, this means the same as RESELECT. If there was not a timer in the future, this means that the prediction phase is over. Go to point 7.

   UNCERTAIN means that an uncertain, unexpandable process was found. Go to point 7.

OVERFLOW means that timing requirements could not be satisfied, and deadlines are missed for some tasks. Go to point 7.

7. Plan execution by running the planning algorithm described in section 5.3.

8. Execute some part of the planned schedule. If less than the entire schedule is executed, this gives the scheduler a chance to take unexpected timing variations into account. Then return to point 2.

5.2 Serialization Algorithms

The default serialization algorithm adds a process to the serialized list. It then returns CONTINUE to the select algorithm, or UNCERTAIN, if the next process cannot be determined without running the program up to that point. Uncertainty will be described in section 6.1.3. There are specialized serialization algorithms for certain types of processes.

5.2.1 Serializing TIME

When serializing a `TIME` block, a timer is created and added to the timer list. RESELECT is returned. When serializing the end of a `TIME` block, the corresponding timer is removed from the timer list, and discovery is done on the processes that follow the timer block. If a timer is discovered following another timer, it will have its earliest permitted deadline set to the time when the first timer expires.

If a timer block ends later than its deadline, a deadline is missed, and OVERFLOW is returned. This leads to the planning phase of the scheduler, where tasks can be prioritized or removed to deal with the missed deadline.

5.2.2 Serializing Send/Receive

All channels have sender-owners and receiver-owners. The owners are the processes which may eventually lead to a send or receive on that channel. Communication processes also have two preceding processes (the processes preceding the send and receive), and two following processes (the processes after the send and after the receive). When a communication is attempted serialized, it will instead serialize the owner of the other side of the channel and return CONTINUE. When the sender-owner and the receiver-owner meets, then RESELECT will be returned so that the select algorithm can choose which of the following paths to continue serializing.

5.2.3 Serializing PAR and PAR FOR

When the base PAR is serialized, it runs discover on all children, so that timers directly following the PAR may be activated immediately. Then it returns RESELECT. This has two purposes. One is that if the serialization is driven by a channel, then only one branch of the PAR may need to be serialized, and the select algorithm will correctly choose that branch. The other reason is that a new timer may have been detected during discovery.

If there is no particular branch that needs serialization, then the entire PAR must be serialized. One branch will first be chosen randomly. Then, when the end of the PAR is to be serialized, it will choose another random branch. Only when all branches are serialized, will the end of the PAR serialize the process following the PAR block.

5.2.4 Serializing ALT

When serializing an ALT, an alternative will be chosen as next, and CONTINUE is returned. An algorithm which chooses the optimal alternative to take in a general ALT is not possible to implement in a scheduler. The different branches of an ALT may be connected to different timing requirements, and the actions connected to each alternative may have different computation times. This makes the problem equivalent to the knapsack problem, which is known to be NP-hard.

However, there are special cases in which the optimal choice is clear. In TIME/occam, there must be a timing requirement associated with some process that leads the ALT to be serialized. The process which forces serialization of the ALT must also be the process with the earliest deadline. If serialization of the ALT is driven by serializing the other end of one of the channel alternatives, then that alternative should be chosen, because it would keep to the EDF principles. If a channel communication alternative is ready (other side is ready to send), then that alternative must be the driving force behind the serialization of the ALT, and that alternative will be chosen.

If it is the ALT itself that is in a `TIME` block, then that timing requirement will be satisfied fastest by just skipping the ALT, if SKIP is an open alternative. The TIME/occam scheduler will therefore always select SKIP, if there are no alternatives that are ready.

The other cases are what make the general problem NP-hard. One case is that a channel communication is driving the serialization, but that alternative is behind a closed guard. Another scenario is that it is the ALT itself that lies in a TIME block, but there is no SKIP. These other cases are handled by picking a random alternative. The order of alternatives, from best to worst, is:

- Channel in open guard where sender is ready

- SKIP
- Any open channel

There is one special case that is not properly handled by random choice, and that is if a channel is driving serialization of the ALT, but that channel communicates in an action, and not in an alternative. An example follows, where the ALT serialization is driven by communication on `ch1`, and the alternative `ch3` is the best choice. As the scheduler is designed now, this would not be detected.

```
PAR:
    TIME 10:
        ch1 ! x
    ALT:
        ch2 ? var:
            SKIP
        ch3 ? var:
            ch1 ? var
```

5.3 Planning Algorithms

After the program is serialized, it can be analyzed for missed deadlines or slack. In the first case, the missed deadlines can be dealt with by either prioritizing or dropping timers. In the latter case, slack can be removed by slowing down or inserting pauses. This phase of the scheduler has not been implemented.

5.3.1 Dealing with Missed Deadlines

Section 4.2 discussed different ways to deal with missed deadlines. Priorities could be used, so that the most urgent tasks get to finish before less urgent tasks. Note again, that priorities in this context only matter if deadlines are known to be missed. Another action is to remove an entire timing requirement; either always with a `SKIP`, or if it would satisfy some weakly hard timing requirement.

In either case, the original serialized schedule cannot be used. The priorities and skips must be activated, and the serialization must be done over again. This may incur a considerable overhead.

5.3.2 Dealing with Slack

If, at any point, all timers in the timer list have start times in the future, we have slack. There are two ways to handle this slack. The first is to pause, and the second is to slow down. The latter, if properly implemented, would save power, as discussed in 1.7.

With the serialized list of processes, the “child” attributes connecting dependent processes to each others, and the timer starts and deadlines; all information is available for slowing down execution without breaking deadlines.

5.3.3 Uncertainties

If there is an uncertain process in the serialization, then processes must be executed up to the uncertain point, before estimation can continue. If there are timers that have their requirements guaranteed, then they can be serialized normally, but the rest must be serialized *as if timing requirements will be missed.* When the uncertainty is removed, the prediction phase can continue as normal.

Part III

Implementation

Chapter 6

System Data Structures

A native program is compiled into a set of instructions. A TIME/occam program must also store information on timing, channel ownership, and parallelism. In particular, the scheduler must be able to trace future channel communications at any point, and to determine the execution path of future execution. The program parts must also be parametric, so that it is possible to instantiate the same program code with different arguments.

This latter requirement was fulfilled by using classes to store the program. A TIME/occam program is compiled to a set of C++ classes with methods. These, along with the scheduler and system classes are compiled from C++ to a self-contained executable program.

6.1 Program State

The scheduler needs to hold state information for all active processes.

6.1.1 Variables

A normal operating system scheduler accomplishes this by storing the processor state, which includes the stack pointer. The stack pointer, along with information on placement of local variables, is enough to allow a thread access to its own locals after a context switch. An OS scheduler also needs to store the CPU state, because it interrupts threads at entirely random points in the execution.

The TIME/occam scheduler does not need to store the CPU state, but it needs to keep track of local variables. In TIME/occam there is also the added difficulty of several parallel threads existing in the same scope. Also, if a variable is changed during a `PAR`, the value must be retained after the parallel threads join, as in the following example:

```
PROC Proc():
    INT [2] x
```

```
SEQ:
    PAR:
        x[1] = 1
        x[2] = 2
    PRINT x[1], x[2]
```

The simplest way to accomplish this transfer of values is to have a single memory area for each variable in each scope, and pass the pointer around. Even better would be to have a `class` or `struct` for each scope, and pass a single scope struct around, but in the implementation, each variable is passed on by its own pointer.

There is a catch to this, however. Because loops may be expanded before execution, iterators in for instance `FOR` loops will change before execution starts, and can therefore not be passed by reference. For example, say that a `FOR` loop iterates variable `i` from 0 to 10. If the entire loop is expanded in the planning phase, then the variable `i` must have different values for each of the iterations. If a single pointer is used this is not possible. Loop iterators are therefore passed by value. It is never possible to pass a loop iterator or `VAL` variable on by reference.

Memory for the variable can be released at the end of scope. This may intuitively sound dangerous if the pointer is passed on to other processes, but because all processes are properly nested, no pointer to a variable will exist beyond end of scope. Any processes or procedure calls must necessarily exit before the calling process exits.

In the implementation, every statement is a C++ class that has a member pointer for every local variable it has access to. Variables that are inherited from previous processes are passed on through the class constructor.

6.1.2 Pop and Children

In a processor, the register that holds the next instruction to be executed is called the "program counter". This is the value that an OS scheduler stores away so that a thread continues where it left off after a context switch. The "link register" is used when calling functions, and is the next instruction to execute after the called function exits.

In TIME/occam, these two variables are called *child* and *pop*. Most processes only have one child, which is the next statement to execute. Only `PAR` statements have multiple children. If there are no more statements in that level in the call stack, then it will "pop", and choose the next statement from a lower level of the call stack.

The variable "pop" is given by the parent process. A process will inherit the "pop" of its parents, if the parent did not explicitly give another. A `PAR`

#	Code	Child(ren)	Pop
1:	`PROC Proc():`	2	Next in calling
2:	`    INT x`	3	Same as 1
3:	`    SEQ:`	4	Same as 2
4:	`        PRINT x`	5	Same as 3
5:	`        PAR:`	6, 7	Same as 4
6:	`            Proc2()`	-	10
7:	`            SEQ:`	8	10
8:	`                Proc3()`	9	Same as 7
9:	`                PRINT x`	-	Same as 8
10:	`        PRINT x`	-	Same as 5

Table 6.1: Example of "Children" and "Pop"

statement, for instance, will give all children its next statement as "pop", so that they will know where to join.

An example of "child" and "pop" is given in table 6.1

6.1.3 Expanding and Uncertainty

Expanding a process means to determine its execution time, and to determine its children. In the above example, the children of each process may be determined at compile time, but this is not always the case. The children of `IF`, `SEQ FOR`, `PAR FOR` and `WHILE` statements may be dependent on run-time values. Also, the "pop" process may be dependent on which context the current process was called from.

A `SEQ FOR` with constant bounds has children that can be found compile-time. If the bounds are complex functions, or depend on values that need to be calculated first, it may not be possible to find the children until all processes up to it have run. Likewise with execution time; there are cases where the execution time cannot be found, or cannot be found until parent processes have run. In these cases the process is labelled *uncertain*. Uncertain processes cause an end to the prediction phase of the scheduler: It cannot predict timing beyond an uncertain process.

There are also cases in between, where children can be found after expanding the parents, but not at compile-time; like an `IF` with an expression that depends on an iterator. These processes are not uncertain. It is a task of the compiler to flag the correct processes as uncertain.

6.2 Base Classes

The `Proc` class holds general state information for processes, including "pop" and "child" variables. The `Proc` class is an abstract base class that must be subclassed to be used. `Proc`, and the direct subclasses to the `Proc` class are pre-written, and a part of the scheduler system.

Each of the subclasses is again subclassed by a class specific to each use. The specific class holds the variables in scope for that process, and definitions of virtual functions specific for that process. These subclasses are created by the TIME/occam compiler, and contain in sum the entire TIME/occam program.

6.2.1 Virtual Functions

The specific classes, created by the compiler, communicate with the system by calling system functions, and replacing virtual functions from the base classes.

`connect()`: This function sets a child process. The default version generates a run-time debugging error if more than one child is attempted connected. It is replaced by the `ParProc` base class, which allows for more than one child.

`expand()`: This function uses `connect()` to add children. It is replaced by the specific classes to connect the actual next process.

`run()`: Code to be executed goes here. Actual execution is distributed among these functions.

`getExecTime()`: Used with `run()`. Should return a safe execution time estimate of the code that is executed in `run()`.

`isUncertain()`: Must return `true` for uncertain processes, and `false` for others, as explained in detail in section 6.1.3.

`isDiscovery()`: Must return `true` for processes that should be expanded even if they have no timing requirements, and `false` for others.

`transfer()`: Technical: Used by communication processes to a call C++ template function. Must be created to call `transfer` on the correct channel.

Figure 6.1 shows base classes for different types of processes, and which virtual functions that are replaced.

Example

For example, each instance of an `IF` inherits from `ControlProc`. It must overwrite the `isDiscovery()` function to return false, to signal that it is not a discovery function. This applies to all instances of `IF`. It will also have a unique `expand()` function that contain the actual conditional expressions

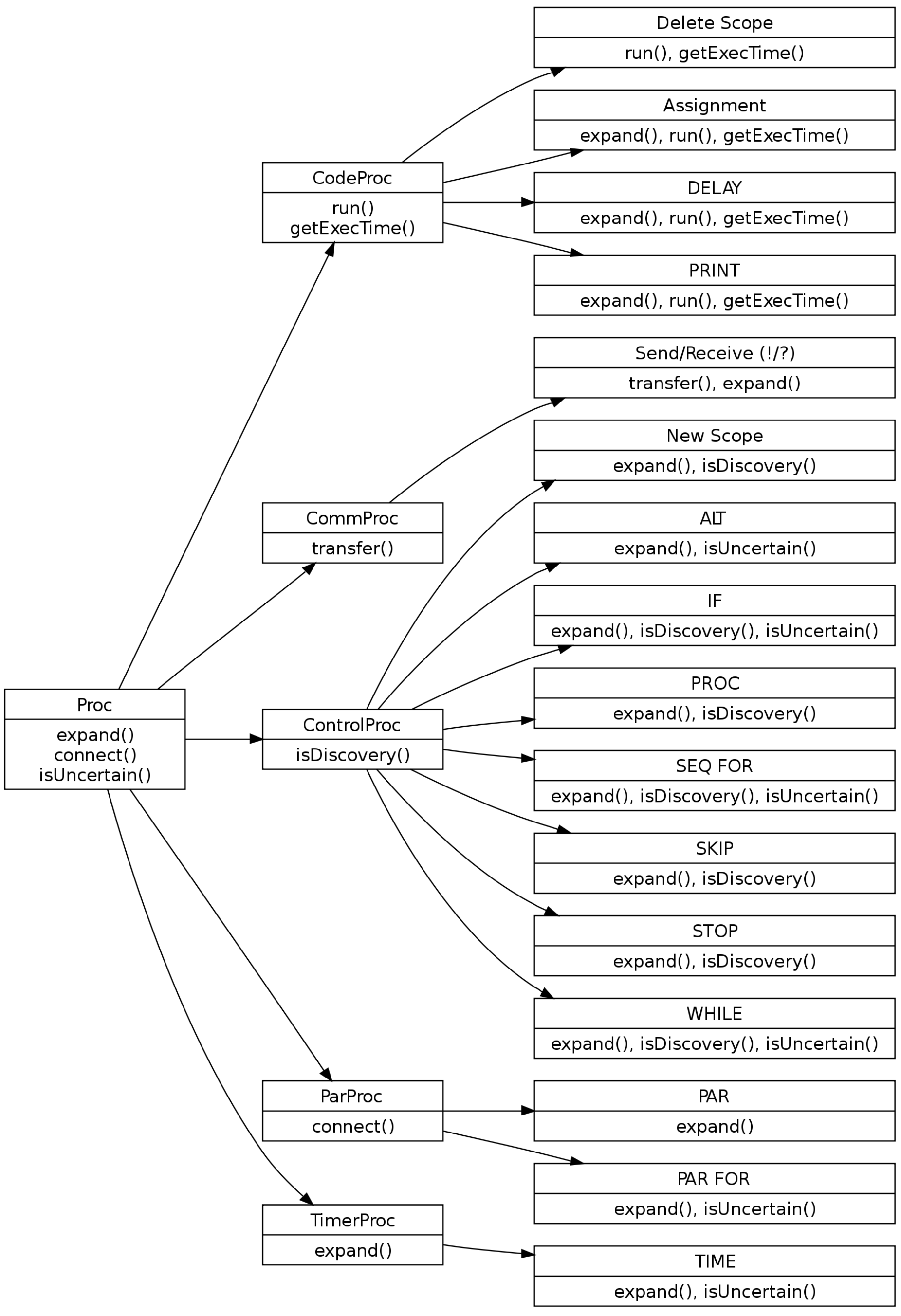

Figure 6.1: Class inheritance diagram with virtual functions

of that `IF`. Finally, it must overwrite the `isUncertain()` function to return whether or not the child can be found before parent processes have executed. This depends on the conditional expression. If it is simple enough, and only depends on iterator variables or constants, it is not uncertain.

6.2.2 CodeProc

This base class has the `run()` and `getExecTime()` functions. The first contains the actual code to be executed when this process runs. The second contains the timing estimate for that code.

6.2.3 CommProc

This base class has a `transfer()` function. This always calls the `transfer()` function on the channel variable that is used for communication. It must be overloaded because the channels are implemented as C++ templates, while the base system (except for the `Channel` class) is not.

6.2.4 ControlProc

`ControlProc` is the only base class where some specific children are discovery processes, and some are not. `IF` is not a discovery function, for example, but `WHILE` is. The `isDiscovery()` function must be made to reflect this.

6.2.5 ParProc

The parallel processes (`PAR` and `PAR FOR`) are the only processes that are allowed to have more than one child. This class contains a new member that holds a vector of children instead of just one, and a new `connect()` function that uses it.

6.2.6 TimerProc

`TimerProc`s does not behave like normal processes. Instead of a child, they have a "next", which specifies the next process in the time block that has not yet been serialized.

6.3 Visualization with Graphviz

In order to help debugging and to visually illustrate concepts, the scheduler can output visual graphs of all the processes in the system and their process dependencies and states. The graphs are output in DOT format, and can be processed by a utility called `graphviz` [27] to generate images.

`graphviz` takes a text file with a list of edges between nodes, each specified as "`Node1 -> Node2`". It then creates a directed graph. If any node names are equal, they are assumed to belong to the same node. `graphviz` has a long list of visual options on how to draw the nodes and edges. In this document, several figures have been created with `graphviz`, including figure 1.6, 1.7, 3.1, 3.2 and 6.1.

6.3.1 Legend

Figures 8.2 to 8.5 in pages 107 to 111 show part of a program state graph. Each node is a process.

- The first line in each node is their C++ class name.
- The second line is a description, written by the compiler. This is typically what kind of statement the process represents.
- The number in brackets is the index of the process. The first process created by the program is process 1. Every time a process is made the index is incremented. The index is only used in debugging, to show the order in which processes are created, and to be able to compare the graph with data from `printfs` or from a debugger.
- Processes with an associated execution time are marked "ET:x".
- Finally, there is a single letter describing the state of the process:

`+`	The process has not been expanded.
`-`	The process has been expanded, but not serialized.
`S`	The process has been serialized.
`X`	The process can be deleted from memory.

The pop class is marked by a dashed line labelled "pop". Dotted lines labelled "depends on" means that a process is waiting for a channel owned by the other process. Timing requirements are shown as yellow triangles. Start and end of timer block are marked by matched triangles, and a dotted line labelled "pair".

Serialized processes are shown in green, uncertain processes in red and timers in yellow until they are serialized. Deletable processes are gray, other processes are white. Parallel branches are simply a single process node with several children edges. Communications are shown as nodes pointing to each other.

6.4 Process Implementation

The compiler uses the base classes as it compiles a TIME/occam program into a set of C++ classes. An example of such a class is listed in figure 6.2. The class is part of the dining philosophers example from chapter 8. This section explains briefly how each TIME/occam process is implemented.

```
// {{{ Class Main_1_1 : ParProc (PAR FOR)
class Main_1_1 : public ParProc {
    private:
        const int* n;
        ChannelVector<int>* chans;
    public:
        virtual void expand();
        virtual bool isDiscovery() const {return false;}
        virtual std::string getName() const {return "Main_1_1\\nPAR FOR";}
        Main_1_1(Proc* pop, const int* P_n, ChannelVector<int>* P_chans) :
            ParProc(pop),
            n(P_n),
            chans(P_chans) {}
};
// }}}

// {{{ Function ‘void Main_1_1::expand()’ (PAR FOR)
void Main_1_1::expand() {
    ParEndProc* join = add(new ParEndProc(pop, this));
    int i;
    for (i = 0; i <= ((*n)-1); (i)++) {
        connect(add(new Main_1_1_1(join, n, chans, i)));
    }
}
// }}}
```

Figure 6.2: Example of C++ class implementation of PAR FOR

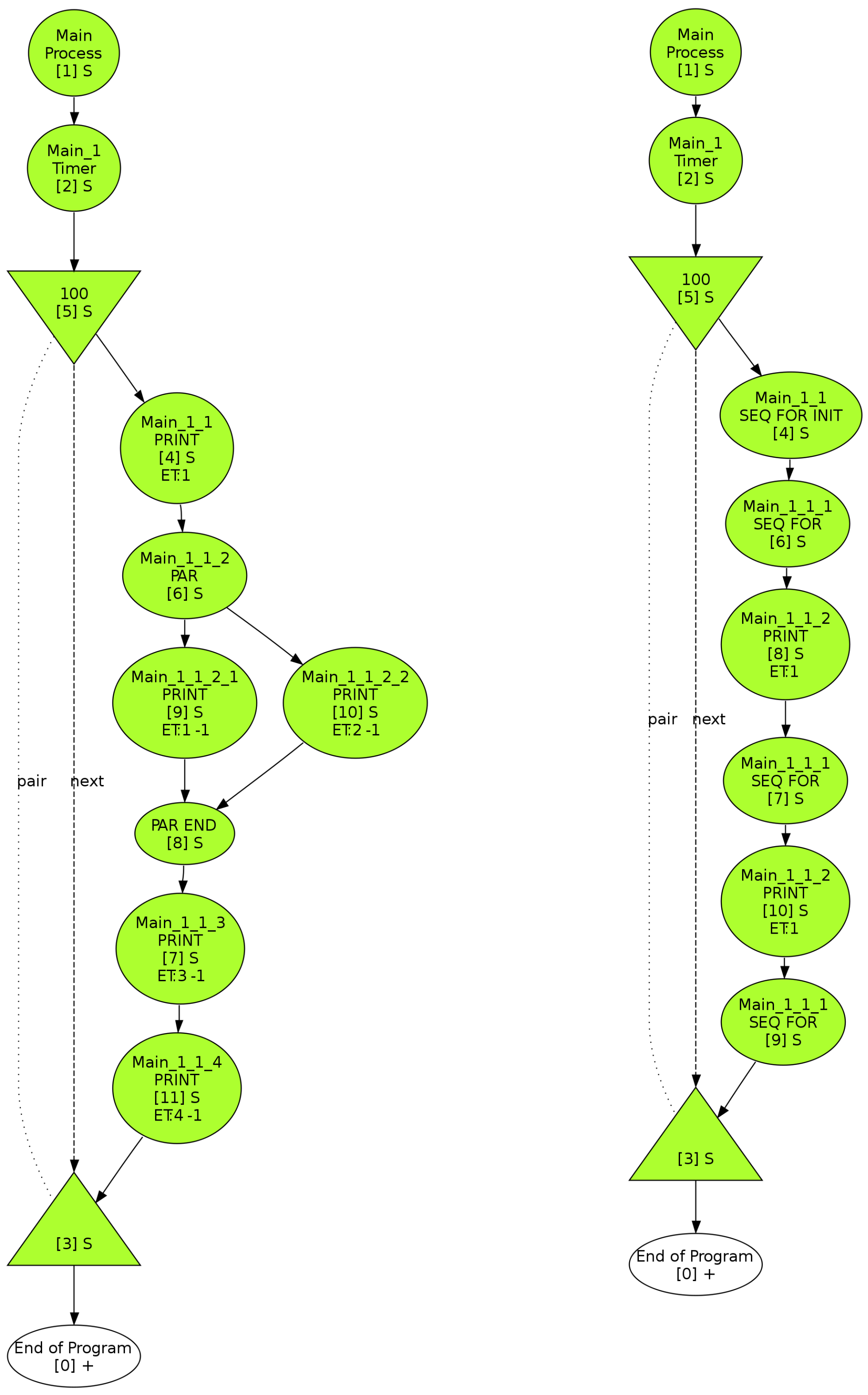

Figure 6.3: Left: Example of SEQ and PAR. Right: Example of SEQ FOR.

STOP

STOP is implemented as `ControlProc`. The expand function should be empty, replacing the default expand function without adding any children. When the scheduler finds an expanded process with no children, it will interpret it as STOP.

SKIP

SKIP is implemented as `ControlProc`. The expand function should connect to next process, if any.

ALT

ALT is implemented as a subclass to `ControlProc`. The expand function must be on a special form, and is a bit complex. A loop iterates through all alternatives three times.

1. The first time, if it encounters an alternative with an open guard, and a channel that has sender ready, the alternative is created and connected to. `expand()` then returns.

2. The second time, if it encounters a SKIP with an open guard, it connects to `pop`, and then returns.

3. The third time it connects to any alternatives with an open guard.

If no alternatives are attempted connected, the process will have no children and it will behave as STOP. The alternatives of an ALT block are not shown in the program state diagrams; because the alternative is resolved before the process connects to the children.

DELAY, PRINT

These processes are implemented as `CodeProcs`. They expand to the next statement in sequence if any; otherwise they default to connect to “pop”. The `run()` function contains the code for the actual action.

IF

The subclassed expand function should evaluate the conditionals, and connect to the appropriate child when a conditional is true. If there is no `else`, a default `else` is added that connects to “next” or “pop”.

SEQ, PAR and PAR FOR

`SEQ` processes are implemented by setting the “next” attribute to the next process in the sequence, and do not have their own control process.

A `PAR` is simply a process with several children. It is a subclass of the `ParProc` super class, which allows more than one child. The expand function should create a `ParEndProc`. Then it should connect to all children, passing the `ParEndProc` as “pop”.

The left graph in figure 6.3 shows the use of classes to represent a program with SEQ and PAR. The source code is the following:

```
PROC Main()
    TIME 100:
        SEQ:
            PRINT "Hello, "
            PAR:
                PRINT "cruel "
                PRINT "world!"
            PRINT " ..and"
            PRINT " goodbye!"
```

PROC

Procedures are transparent, and are implemented as the first process in the procedure. Arguments are passed from the caller through the class constructor, just as variables are usually passed from a process to another.

SEQ, SEQ FOR and WHILE

`SEQ FOR` is implemented as a control process that tests and increments the iterator variable, and then calls the block process with a new copy of itself as “pop”, until the loop ends. `WHILE` is implemented similarly to FOR.

The right graph in figure 6.3 shows a SEQ FOR. The loop has only two iterations; otherwise it would be too long to print. The code is the following:

```
PROC Main()
    TIME 100:
        SEQ FOR i = 0 TO 1:
            PRINT i
```

TIME

Time is implemented by wrapping the timed block into a pair of `TimerProc` and `TimerEndProc` classes, so that the first becomes the parent of the block, and the latter becomes child of the last process in the block. This is done by creating a `TimerEndProc` and using it as “pop” for the block. Then a

`TimerProc` is created, with the block as its "pop". Both graphs in figure 6.3 have TIME statements (otherwise the programs would not start).

Chapter 7

The Compiler

To be able to experiment with the TIME/occam language, a compiler was made that translated TIME/occam code to C++ code. The compiler was written in Haskell, with the help of the compiler creation tool `bnfc` [28].

7.1 Compiler Basics

A compiler is a computer program that translates source code from one programming language to another or from one programming language to machine code. To do this, the compiler interprets the source code according to the specification of the source language.

7.1.1 Parsing algorithms

A compiler generator, or compiler-compiler, or parser generator; is a tool that creates a program that may interpret text, based on a lexical and grammatical specification of that text.

There is often a separate program for lexical analysis, called a *lexer*. The lexer breaks the text into symbols, and determines the type of each symbol. The lexer is also responsible for removing comments.

The lexer feeds symbols to the *parser*, which uses its knowledge of the grammar to interpret the text. The parser may execute code on the fly, whenever a rule is resolved, or it may build up a full concrete syntax tree to be used for later.

Most parser generators use an algorithm called LALR(1) (Look Ahead Left-right Rightmost derivation, 1 token look ahead). It cannot parse all kinds of unambiguous grammar, but is efficient.

That it is left-right, means that it parses the tokens from left to right. The rightmost derivation means that it is always the rightmost non-terminal that is attempted to be replaced first. The derivation does not matter if the parser is unambiguous, and if a concrete syntax tree is created before executing code. On the other hand, if the grammar is ambiguous, the derivation

Next token	**Action**	**Stack (after)**
2	shift	2
$\times$	shift	2,$\times$
3	shift	2,$\times$,3
$+$	reduce	6
	shift	6,+
4	shift	6,+,4
$\times$	shift	6,+,4,$\times$
5	shift	6,+,4,$\times$,5
	reduce	6,+,20
	reduce	26

Table 7.1: Example of shifting and reducing: Parsing $2 \times 3 + 4 \times 5$

determines which rules will be applied. Also if code is executed on the fly during parsing, the derivation direction determines what code will be executed first.

Another common algorithm is the GLR (Generalized Left-Right). The big difference between LALR(1) and GLR is that the GLR algorithm never resolves ambiguities: If an ambiguity is encountered, the parser branches and evaluates all possibilities. If a branch later fails, then that branch is removed. If branches meet (end up with identical syntax trees), then they are merged. A GLR parser can parse all kinds of unambiguous grammar. Furthermore, if the grammar is ambiguous, it finds all possible syntax trees.

A GLR parser is slower than a LALR(1) parser. Also, known ambiguities that in a LR(1) is handled in a predictable way (like the dangling `else`, described in the next section) are not resolved in a GLR parser, but rather results in several CSTs.

Shifting and Reducing

The parser analyze the sequence of tokens in a process of *shifting* and *reducing*. Shifting means to push a new token to a stack before continuing. Reducing means to process elements already on the stack. For example: Parsing $2 \times 3 + 4 \times 5$ according to standard mathematical rules yields the sequence of shifting and reducing listed in table 7.1.

A common ambiguity in a grammar is a *shift/reduce conflict* in which the parser does not know whether to shift or to reduce. Writing mathematical parsers without associativity or precedence yields such conflicts. In such cases, the conflicts can be solved by the methods described in section 3.2.4.

Another shift/reduce conflict that is harder to solve, comes with the following grammar, describing C-like `if` statements:

```
<stmt> ::= "if" "(" <exp> ")" <stmt>
```

```
         | "if" "(" <exp> ")" <stmt> "else" <stmt>
         | <action>
```

The problem comes with nested `if`s. If there are more `if`s than `else`s, to which `else` do the `if` belong? This is known as the "dangling else"-problem, and is not easily solved. Most parsers use rightmost derivation, which means that they will always prefer shifting to reducing. The `else` will then belong to the innermost `if`. In the following example, that means `action2()` will be executed if `x` and not `y`.

```
if (x) if (y) action1(); else action2();
```

The grammar in this case is ambiguous, but it will always be resolved in a predictable way by LR-parsers. Shift/reduce conflicts like this may therefore sometimes be ignored.

Another conflict is the *reduce/reduce* conflict. A simple way to create a reduce/reduce conflict is to have two equal production rules with different results, for instance:

```
<hexnum>   ::= "0x" <hexdigits>
<intnum>   ::= <int>
<floatnum> ::= <float>
             | <int>

<num>      ::= <hexnum>
             | <intnum>
             | <floatnum>
```

Here, a <floatnum> may be both an integer and a floating point number. This may be intentional; it may be required that integers are legal `<floatnum>`s at some other point in the grammar. The problem comes from the definition of `<num>`. If an `<int>` is encountered when needing a `<num>`, the parser will not know whether or not to reduce it through `<floatnum>` or through `<intnum>`. This is an example of a reduce/reduce conflict. In contrast to shift/reduce errors, no simple rule can be used to standardize the parsers behaviour. A reduce/reduce conflict is not a conflict that may be ignored, and is usually a sign that something is very wrong with the grammar.

Problems with One Token Look Ahead

A LALR(1) parser has only got a single token look ahead, before it chooses what to shift or reduce. Having a single token look ahead may make grammar seem ambiguous that really is not. The following example is from the GNU Bison manual [29]:

In Pascal variable types may be defined, amongst other possibilities, as subranges or enums. Examples are:

```
type enum = (a, b, c);
type subrange = lo .. hi;
```

In Extended Pascal, the range limits (`lo` and `hi`) may be arbitrary expressions. If we create a single-element enum, and use that as lower bound of the subrange, we get:

```
type enum = (a);
type subrange = (a) .. hi;
```

This cannot be unambiguously parsed with only a single token look ahead. To determine whether the type is really an enum or a subrange, the parser needs to find the dots (`..`), but this requires to look ahead of all `(a)`, which is three tokens.

Lexer Feedback

The communication between the lexer and the parser is not always unidirectional. Consider trying to parse C declarations and expressions. The asterisk (`*`) is used both as a pointer mark and as a multiplication symbol.

The first of the following statements is a definition of a type, and the second is a pointer:

```
typedef int my_type;
my_type * var_ptr;
```

The statement below is a multiplication. The resulting product is discarded, making the statement rather meaningless. It is still legal, however.

```
my_var1 * my_var2;
```

To separate these, the parser needs to know which identifiers are types, and which are not. This is not known in advance, because the `typedef` keyword may create new types. One solution is to let the parser inform the lexer of new types, creating feedback from the parser to the lexer, in addition to the stream of tokens from the lexer to the parser. (Another option is to process typedefs by a pre-processor.)

7.1.2 Compiler Generator Tools

Lex/Yacc

`lex` and `yacc` are the standard Unix lexer and parser generators for C/C++. On Linux similar programs are `flex` and `bison`. Variants exist for many languages, for instance `Jlex` and `cup` for Java, and `alex` and `happy` for Haskell.

`lex` is a regular expression finite state machine. It matches the text against all patterns available in the current state, and performs actions associated with each pattern. Actions may be to change state, or to output a token. It is the sequence of tokens that are used next by `yacc`.

`yacc` takes a sequence of tokens and matches against a grammar description similar to BNF. With each rule there is an associated action which is performed when the rule is matched. In a programming language parser the action may be to build up a CST; in a calculator the action may be to perform the actual computation on the fly. `yacc` includes commands to simplify grammar files, such as explicit associativity and precedence rules.

bnfc

`bnfc` is a multi-language parser generator that takes a single BNF-like source file, and generates a lexer, parser, pretty-printer and LaTeX language description. It was developed by Michael Pellauer, Markus Forsberg and Aarne Ranta [28]. `bnfc` is part of the software database of the Linux distributions Debian and Ubuntu.

The bnfc source files are in a format called Labelled BNF, or LBNF. It is much like BNF, except that every rule also needs a unique name or label. It does not contain explicit statements of associativity and precedence.

`bnfc` can be used to generate parsers in multiple languages, and uses existing lexer and parser generator tools for each languages. `bnfc` has Haskell as native language, in which case it writes lexer and parser scripts for `alex` and `happy`.

There are several advantages of using `bnfc` instead of using `lex/yacc` variants directly:

- The LBNF grammar file is in itself a complete description of the lexical and grammatical structure of a language. There is not a separate lexer and parser description.
- The LBNF contains nothing else but a description of the language. `lex/yacc`-like source files often become littered with implementation details.
- LBNF contains useful macros, for instance replacing the Alex lexer statement
  ```
  "/*" ([$u # \*] | \* [$u # \/])* ("*")+ "/" ;
  ```
  with the more readable line
  ```
  comment "/*" "*/" ;
  ```
- LBNF is language independent, it produces parsers in C, C++, Java, O'Caml and Haskell from the same grammar file.

```
comment "//";
comment "/*" "*/";

layout ":";
layout toplevel;

entrypoints Program_spec;
```

Figure 7.1: Listing: Tocc.cf, 1: Header

7.2 TIME/occam Grammar Source Code

`bnfc` takes a single input file on LBNF form [26] called the “grammar file“, describing the lexical and grammatical build-up of the language. This section explains some of the contents of the grammar file for TIME/occam.

The LBNF description is not exactly like the BNF description, for two reasons. One, the LBNF description contains lexical definitions, and two, because the compiler is only LALR(1), certain elements had to be rewritten (This is described in section 7.2.5).

7.2.1 Grammar File Heading

The first part of the file contains definitions of comments and layout rules. It is listed in figure 7.1. The first layout line tells `bnfc` that the `":"`-token precedes a new base indentation. The second line tells `bnfc` that there is an implied `":"` preceding the document. With this last line, statements that begin at the left margin are automatically separated by semicolons.

As explained in section 3.3.1, the layout resolver separates statements beginning at the base indentation with semicolons. Any statement following a layout token (`":"`) sets a new base indentation. Any statement with a higher indentation than the base, not following a layout token, is interpreted as a continuation of the previous line.

The `entrypoints` pragma tells `bnfc` only to create compilers for certain program elements. The only entry point here is a complete program.

7.2.2 Lexical Definitions

The next part of the grammar file contains lexical definitions. Many lexical tokens are predefined by `bnfc`, like `Ident` for identifiers, or `String` for quoted strings.

TIME/occam needs to differ between upper-case identifiers (types, keywords), capitalized identifiers (parametric processes), and lower-case identifiers (variables), as explained in 3.3.2. These lexical definitions of these

```
token CapIdent upper (upper | digit | '_')* lower (letter | digit | '_')*;
token LowerIdent lower (letter | digit | '_')*;
token UpperIdent upper (upper | digit | '_')*;
```

Figure 7.2: Listing: Tocc.cf, 2: Some lexical definitions

```
InitExp.      Init_spec ::= "=" Exp1;
InitNone.     Init_spec ::=;

Decl.         Decl_spec ::= Qualifier_spec Type_spec Array_spec LowerIdent;

ArraySized.   Array_spec ::= "[" Exp1 "]";
ArrayUnknown. Array_spec ::= "[" "]";
ArrayNone.    Array_spec ::=;

Var.          Variable_spec ::= Decl_spec Init_spec;
terminator Variable_spec ";";

Param.        Param_spec ::= Decl_spec;
separator Param_spec ",";
```

Figure 7.3: Listing: Tocc.cf, 3: Definitions of declarations

token types are defined in the next section of the grammar file, and are listed in 7.2. Integer and floating-point numbers are also lexically defined in the grammar file, but are not listed here.

7.2.3 Declarations

The declarations part describes variable and parameter declarations, types and qualifiers. The `terminator` and `separator` macros are used to define how to handle lists of the specified type. A variable declaration list, is a set of variable declarations *terminated* by "`;`" (following each element), while a set of parameter declarations is a list of declarations *separated* by "`,`". Later uses of the expressions `[Variable_spec]` and `[Param_spec]` will mean such lists.

7.2.4 Processes

The definitions of processes are generally straight forward. Assignments are processes, and not expressions, like in for instance C. Variants of the C statement "`if ((x = malloc(y)) == NULL) ...`" is therefore not legal.

The difference between a `SingleStBlock` and a `Block` is that the first

```
Block.          Block_spec ::= "{" [Variable_spec] [Proc_spec] "}";

SingleStBlock.  SingleStBlock_spec ::= "{" [Variable_spec] [Proc_spec] "}";

OrderSeq.       Order_spec ::= "SEQ";
OrderPar.       Order_spec ::= "PAR";

ProcAlt.        Proc_spec ::= Alt_spec;
ProcAssign.     Proc_spec ::= LowerIdent Assignment_op Exp1;
ProcAssignArr.  Proc_spec ::= LowerIdent "[" Exp1 "]" Assignment_op Exp1;
ProcCall.       Proc_spec ::= CapIdent "(" [Arg_spec] ")";
ProcComm.       Proc_spec ::= Comm_spec;
ProcDelay.      Proc_spec ::= "DELAY" Exp1;
ProcElse.       Proc_spec ::= "ELSE" ":"  SingleStBlock_spec;
ProcElseIf.     Proc_spec ::= "ELSE" "IF" Exp1 ":" SingleStBlock_spec;
ProcExp.        Proc_spec ::= Exp1;
ProcFor.        Proc_spec ::= Order_spec "FOR" LowerIdent "=" Exp1
        "TO" Exp1 ":" SingleStBlock_spec;
ProcIf.         Proc_spec ::= "IF" Exp1 ":" SingleStBlock_spec;
ProcOrder.      Proc_spec ::= Order_spec ":" Block_spec;
ProcPrint.      Proc_spec ::= "PRINT" [Exp1];
ProcSkip.       Proc_spec ::= "SKIP";
ProcStop.       Proc_spec ::= "STOP";
ProcTime.       Proc_spec ::= "TIME" Exp1 ":" SingleStBlock_spec;
ProcWhile.      Proc_spec ::= "WHILE" Exp1 ":" SingleStBlock_spec;
separator nonempty Proc_spec ";";
```

Figure 7.4: Listing: Tocc.cf, 4: Definitions of processes

does not allow for more than one statement. The only way to list several statements is to use one of the ordering keywords, `SEQ` or `PAR`. This rule also exists in occam.

However, there is no grammatical difference between a `SingleStBlock` and a `Block`, as their LBNF definition is equal. This is because of the way conditionals are defined:

7.2.5 Why `IF` and `ELSE` are grammatically independent

The statement

```
separator nonempty Proc_spec ";";
```

together with the layout rules means that any set of lines at one base indentation will become separated by semicolons. Trying to use the rules

```
ProcIf.         Proc_spec ::= "IF" Exp1 ":" SingleStBlock_spec Else_spec;
ElseElseNone.   Else_spec ::=;
ElseElseIf.     Else_spec ::= ";" "ELSE" "IF" Exp1 ":" SingleStBlock_spec
        Else_spec;
ElseElse.       Else_spec ::= ";" "ELSE" ":" SingleStBlock_spec;
```

will lead to a shift/reduce conflict because of lack of look ahead. When parsing for an `Else_spec` and encountering a "`;`", the parser does not know with one token look ahead, if that semicolon is part of the `Else_spec`, or the separator between this and the next process statement. The BNF description given in appendix A is created without considering look ahead, so the BNF description and the LBNF implementation differs at this point.

The solution is to make `IF` independent from `ELSE` and `ELSE IF`, as seen in the grammar. This does not mean that having stray `ELSE`s around in the program is semantically meaningful; it only means that this is an error that is not detected during parsing.

A single-statement (unordered) process may be a conditional. Because of the independence of conditionals, such a process may contain multiple `Proc_spec`s. Missing ordering of processes is therefore another programming error that cannot be detected during parsing. Having two different production rules (`SingleStBlock` and `Block`) is still useful, because it eases the implementation of such a check in the compiler.

There are multiple other solutions to the problem of having grammatically independent `IF`s and `ELSE`s: One is not to use the `separator` macro, but implement a specialized list definition. Another is to define different layout rules for conditionals, where only the initial `IF` has base indentation, while consequent `ELSE`s and `ELSE IF`s are further indented. A third option is to use a parsing algorithm with more look ahead.

7.3 TIME/occam Compiler Organization

In addition to the straight-forward translation of expressions, the TIME/occam compiler needs the following features:

- Restrict sharing of variables and channels. Only read/read is allowed on one variable in parallel. Restrict sharing of channels. Only one parallel process may send to and only one process may receive from a channel at any given time.
- Do the same for arrays, when indexing can be guaranteed at compile-time.
- Keep track of uncertain expressions; minimize the number of processes marked uncertain. (An uncertain process is one which cannot be expanded before parents have run.) This has not been implemented.

The compiler is divided into 9 modules:

ToccCompile.hs Compiler. Translates a TIME/occam source into C++ classes. The compiler is described in section 7.3.1.

ToccConstExp.hs Evaluates the value of constant expressions. This allows variables that are known at compile time to be used in expressions, which results are also known at compile time. There are three levels of "knownedness": "Unknown", "known at expand-time" and "known compile-time". If there is a single unknown in an expression, the entire expression returns "unknown". If there are no "unknowns", but a single "known expand-time" the result is "known expand-time". Otherwise the result is calculated and returned "known".

Tocc.hs Main module. Opens files and calls appropriate functions.

ToccLib.hs Contains some common functions.

ToccListVar.hs Lists all variables that a program, program part or program branch uses, and in which ways they are used. Can check a program for shared variable violations. It can also split a scope so that each branch in a `PAR` gets variables that they use, and only variables that they use. (This is important for channels, as branches which do not use a channel must not be registered as owner for that channel). This module is explained further in section 7.3.2.

ToccLocalsDecl.hs Contains definition of the `Local` type. This must be in a separate file to avoid circular dependencies between modules.

ToccLocals.hs Contains functions that manipulate and convert local variables and scopes.

ToccOpt.hs Parses command line options using the `getopt` library.

ToccProto.hs Scans a program for procedures, and determines qualifiers for parameters.

7.3.1 ToccCompile

The concrete syntax tree is output from `bnfc`. The compiler passes through the CST, outputting C++ classes whenever necessary. The values `ps`, `locals`, `name` and `next` are passed along. `ps` is the CST for the entire program. This is used to search for procedure parameter types for procedure calls (`PROC`). `name` is the class name that the next class must have, as it is the type of class that the parent creates to proceed. `next` is the class name that the next class must use as its child, and is used exclusively by `SEQ`/`SEQ FOR`. If no class `next` is provided, a class must connect to pop. `locals` is the complete list of local variables in the current scope.

The functions `tpl*` are template functions that write the main class code. Code is split into type, header and source parts, contained in a `CCode` type. The type part contains typedefs, the header part contains class declarations, and the source part contains class function definitions. Classes will often have circular dependencies in the `expand` functions, so the declaration must be separate from definitions. When `CCode`s are added, the three parts are separately concatenated. Types are output first because they must appear in the C++ code before anything else, then headers because they must appear before the main source, and then the source.

The compiler is otherwise a fairly straight forward conversion between TIME/occam source code and the data structures described in chapter 6.

7.3.2 ToccListVar

The `listvar*` functions returns all variables used from the current scope, by all statements and substatements of a given process. If a new scope defines another variable with the same name as one in the original scope, uses of this variable are removed.

When run on each branch in an `ALT`, the functions lists all uses of existing variables by each branch. The uses can then be merged with the `combineUsesPar` function, which also reports errors if variables are illegally shared.

Array elements are harder to track. Unless legal sharing can be guaranteed, it is an error. If indices are known at compile-time, they are explicitly checked. The listvar module also understands that if an array is indexed by an iterator variable in a `PAR FOR`, then the uses must necessarily be disjunctive.

The `needLocals` function takes a scope and procedure, and returns a set of (param, arg) representing the use of local variables in that procedure.

Param is the use, for instance `CONST`, while arg is the original variable being used.

Chapter 8

Dining Philosophers Example

This chapter will show an implementation of the dining philosophers problem in TIME/occam. The program will be briefly explained, and some steps in the scheduling serialization of the programs will be shown.

Dining philosophers is a good example of general channel use and parallel processing. The source code is shown in figure 8.1. Three philosophers are used. Figures 8.2 and 8.3 show the program state after the initial discover. Note that the `PAR`s, calls and scopes are all expanded during the initial discovery.

Discovery stops at `ALT`, and `PRINT`. All processes below those points are connected by "pop"s; that is, they represent popping the call stack. These are processes that free memory, marked "Delete Scope"; and the parallel join nodes marked "PAR END". There are also the next "WHILE" processes, representing the next iteration of the loops.

Now, the earliest deadline timer is selected. All deadlines are equal, so the first (left) timer is chosen. It will print "Philosopher 1 wants to eat", and then send on the `leftFork` channel, then print "Philosopher 1 has got left fork", and then send on the `rightFork` channel; following the procedure `Phil`.

To send on a channel, the other side must be ready. The other side is a `Fork` process, which loops around an `ALT` without any timing requirements. To communicate, the `Phil` procedure forces the `Fork` procedure to progress into the `ALT`, choosing the correct alternative, because that channel is already waiting. Then the `Phil` procedure prints some more and repeats with the other fork. The program state graph after the first two communications is depicted in figure 8.4.

The program will continue in a similar manner. The entire program state graph is depicted in figure 8.5, but is scaled down to fit. It may not be possible to read all process names, but it still illustrates order of execution.

```
PROC Main()
  CONST INT n = 3
  CHAN INT [2*n] chans
  PAR FOR i = 0 TO n-1:
    PAR:
      Phil(i, chans[2*i], chans[(2*i+2*n-1) % (2*n)])
      Fork(chans[2*i], chans[(2*i+1)])

PROC Fork (IN INT leftPhil, IN INT rightPhil):
  INT dummy
  WHILE 1:
    ALT:
      leftPhil ? dummy:
        leftPhil ? dummy
      rightPhil ? dummy:
        rightPhil ? dummy

PROC Phil (VAL INT philNo, OUT INT leftFork, OUT INT rightFork):
  INT dummy = 0
  TIME 100:
    SEQ:
      PRINT "Philosopher ", philNo, " wants to eat."
      leftFork ! dummy
      PRINT "Philosopher ", philNo, " has got left fork."
      rightFork ! dummy
      PRINT "Philosopher ", philNo, " has got both forks, eating."
      DELAY 10
      rightFork ! dummy
      PRINT "Philosopher ", philNo, " has given away right fork."
      leftFork ! dummy
      PRINT "Philosopher ", philNo, " has no forks."
```

Figure 8.1: Listing: Dining philosophers source code

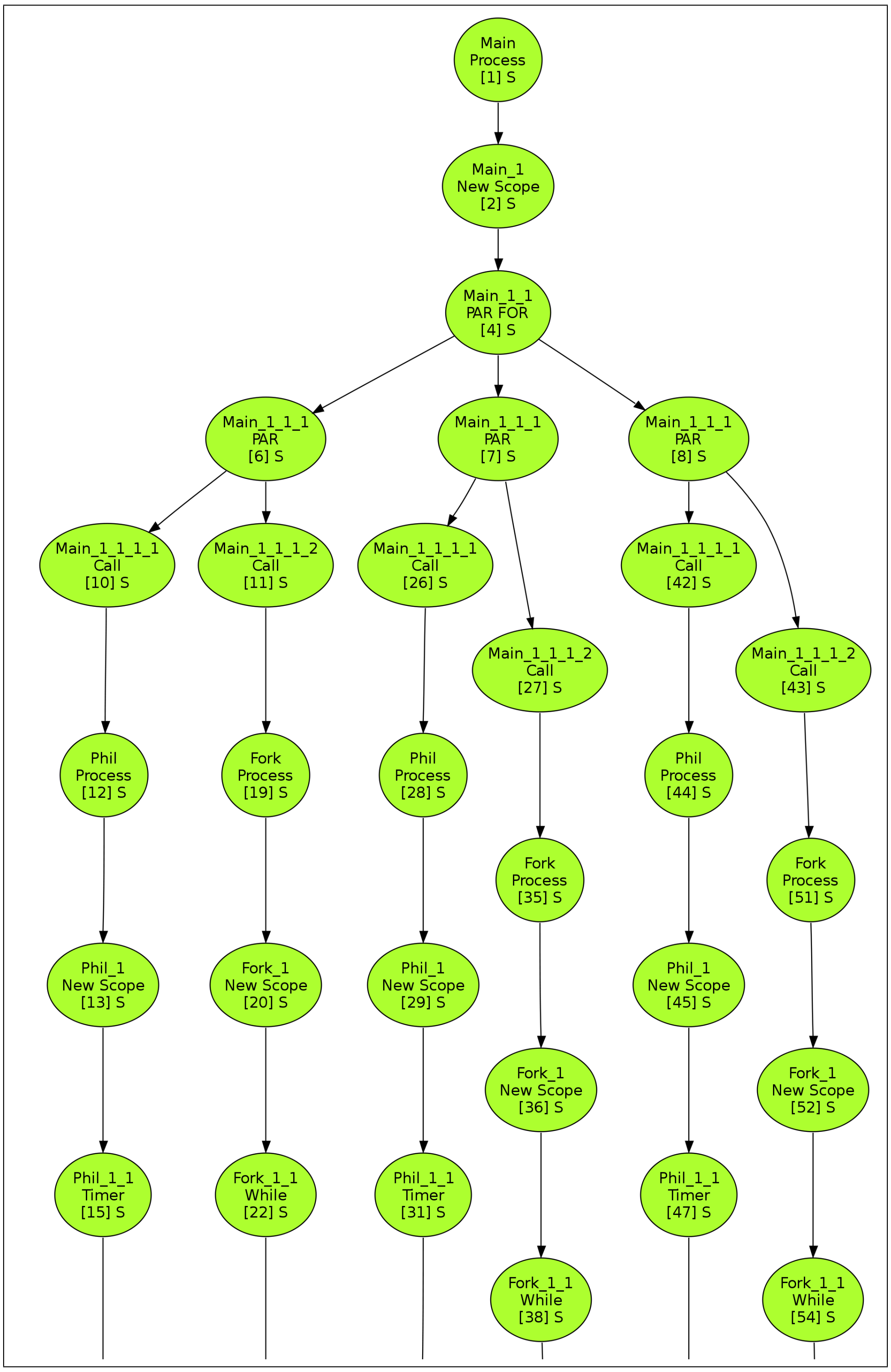

Figure 8.2: Dining philosophers program state graph, after initial discovery, part 1/2.

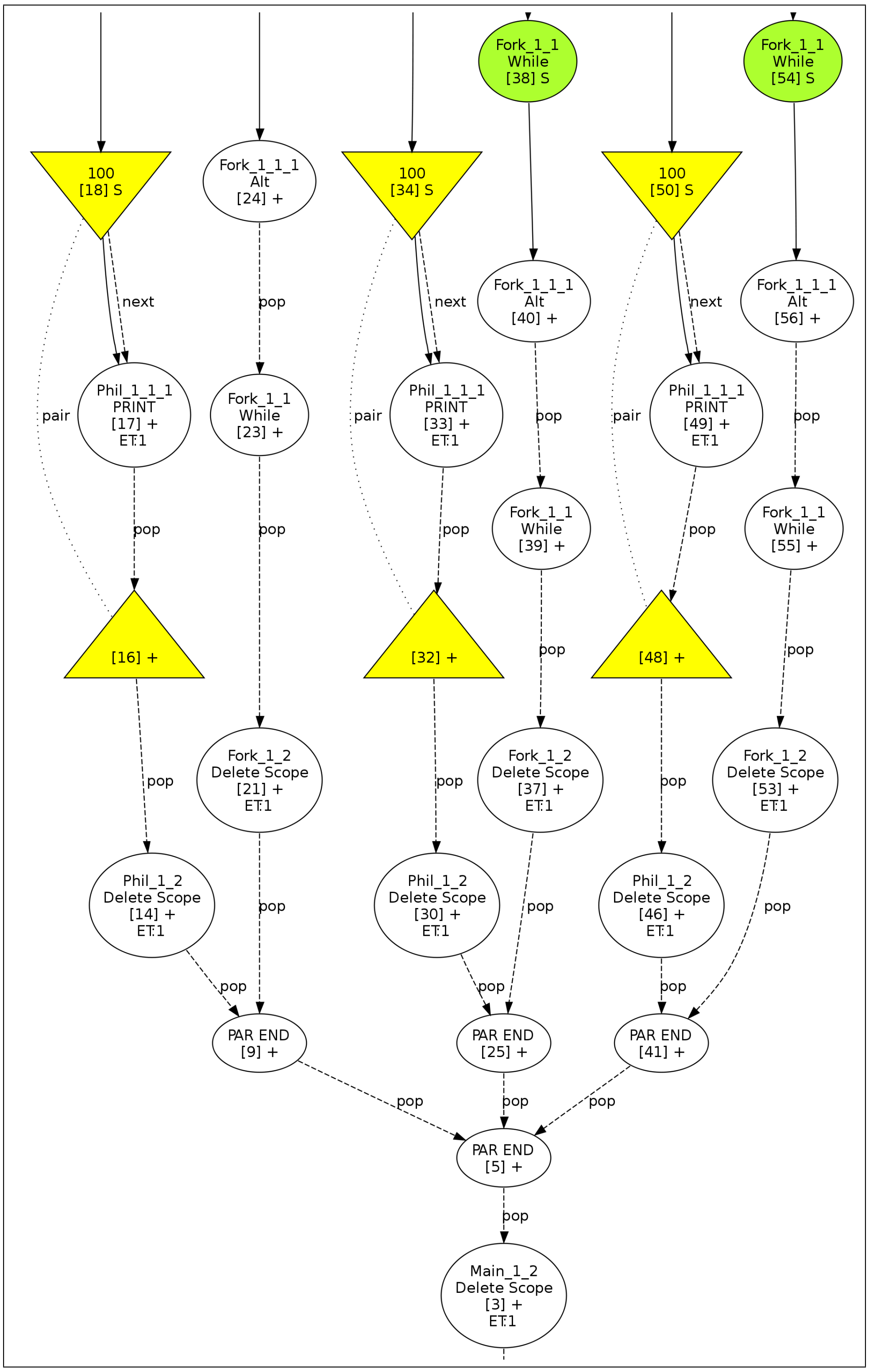
Fork_1_1 While [38] S
Fork_1_1 While [54] S
100 [18] S
Fork_1_1_1 Alt [24] +
100 [34] S
100 [50] S
Fork_1_1_1 Alt [40] +
Fork_1_1_1 Alt [56] +
next
pop
pair
Phil_1_1_1 PRINT [17] + ET:1
Fork_1_1 While [23] +
Phil_1_1_1 PRINT [33] + ET:1
Phil_1_1_1 PRINT [49] + ET:1
Fork_1_1 While [39] +
Fork_1_1 While [55] +
[16] +
[32] +
[48] +
Fork_1_2 Delete Scope [21] + ET:1
Fork_1_2 Delete Scope [37] + ET:1
Fork_1_2 Delete Scope [53] + ET:1
Phil_1_2 Delete Scope [14] + ET:1
Phil_1_2 Delete Scope [30] + ET:1
Phil_1_2 Delete Scope [46] + ET:1
PAR END [9] +
PAR END [25] +
PAR END [41] +
PAR END [5] +
Main_1_2 Delete Scope [3] + ET:1

Figure 8.3: Dining philosophers program state graph, after initial discovery, part 2/2.

Another important point that can be seen out of that figure is that some of the clean-up processes marked “Delete Scope”, are never run, because they are outside of any timing requirements.

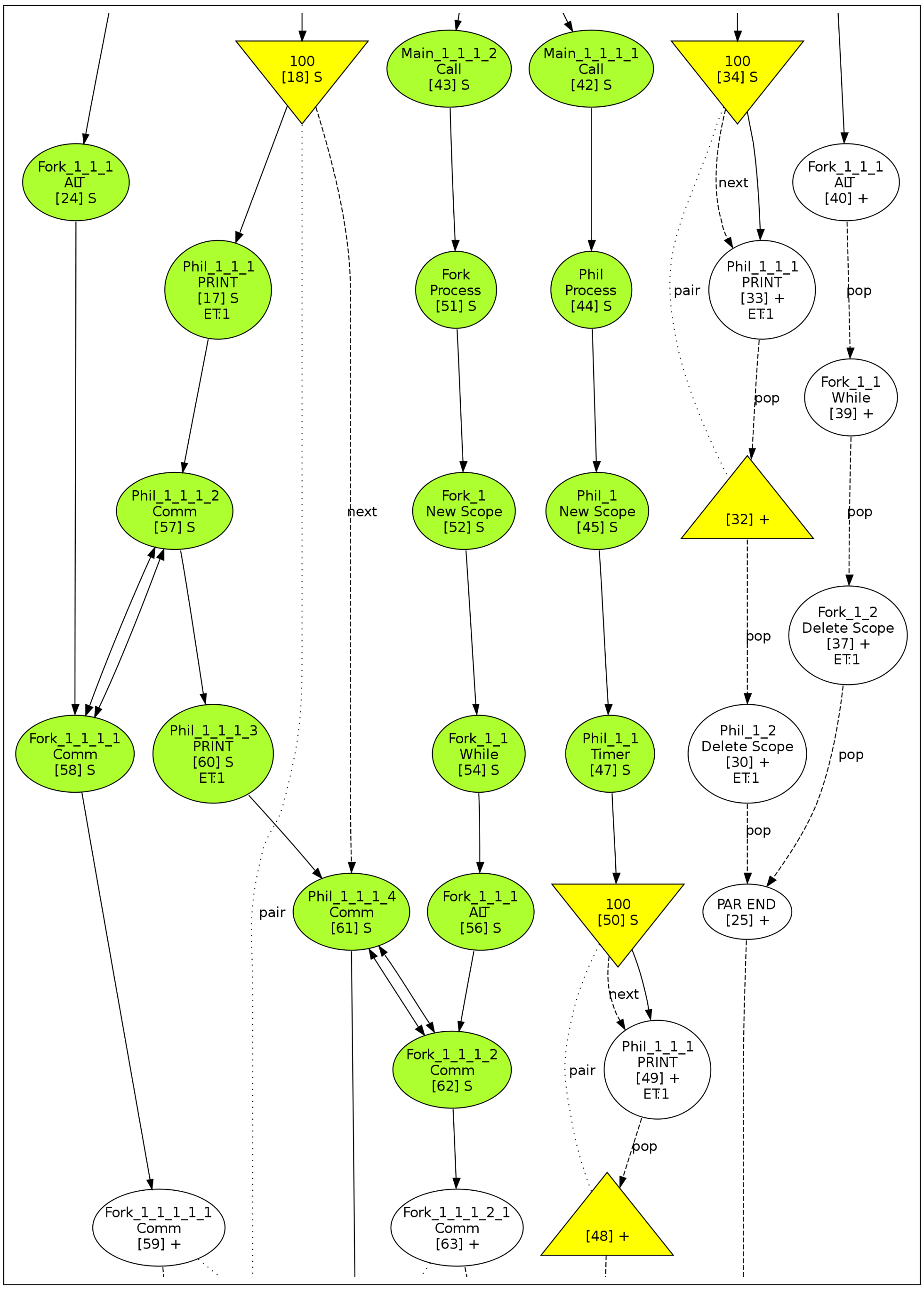

Figure 8.4: Dining philosophers graph, one philosopher eats.

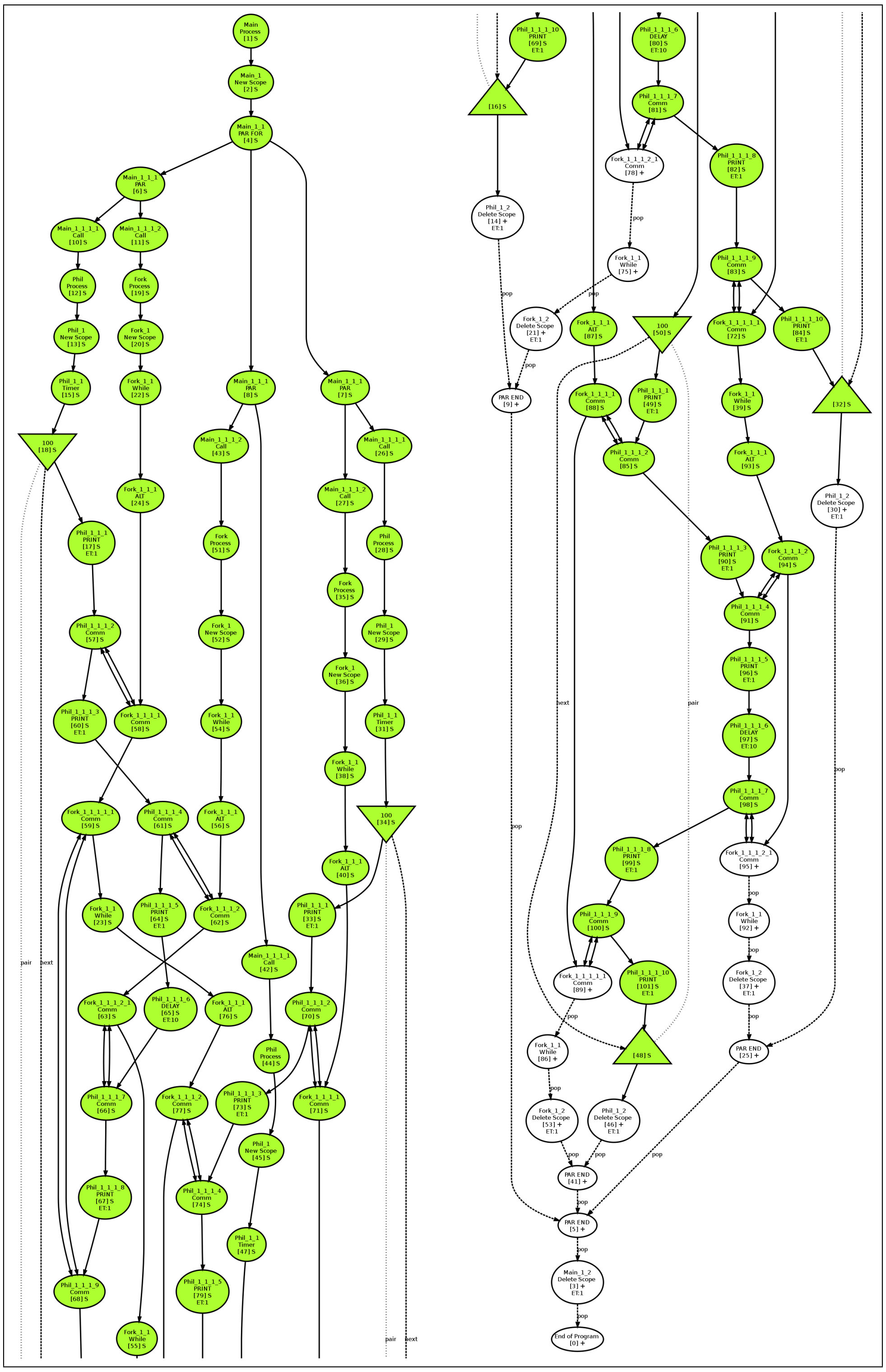

Figure 8.5: Dining philosophers graph, full program.

Chapter 9

Limitations

This chapter describes some of the conceptual limitations of the TIME/occam language. It also lists features that have not been developed or implemented, or features that should be changed. The chapter has three sections; first a list of the conceptual limitations and fundamental problems to be solved; then features of the language design that have not been included, and finally a list of missing features from the implementation of the compiler and scheduler.

9.1 Conceptual Limitations

In theory, the principles of TIME/occam should allow precise control and formulation of real-time constraints from within the language. However, there are a few issues that must be resolved before such a system will work well.

First, there is a question of granularity and overhead. There is overhead attached to traversing the process tree. The larger the processes, the relatively less time is spent on this overhead. Nevertheless, other types of schedulers are concerned with far smaller overheads than this. Even if the scheduler and system is optimized, will such a complex scheduling system ever become feasible?

The system would work best if most programs were predictable. That is, uncertain processes should be rare. If an uncertain process is detected, the scheduler must proceed to it, as if deadlines are to be missed, deviating from the basic optimal EDF scheduling strategy. If priorities, `SKIP`s and weakly hard timing requirements (section 4.2) are specified, they must be used instead if scheduling for all tasks cannot be guaranteed. If programs in general have too many uncertain processes, the principles of TIME/occam may not work well.

9.2 Missing Features in Language

This section contains a list of language features that should be implemented, but have not yet been. The most important missing feature in the language is better support for arrays of channels. There are also multiple minor features that are not implemented.

9.2.1 Arrays of Channels

Better support for arrays of channels is necessary. As of now, elements of channels can only be passed to procedures and not used directly. The reason for this is that the owner of each end of a channel must be known at all times. The owner is the process whose children will eventually communicate on that channel. As of now, all arrays are passed in their entirety, so that either, the channel is completely split into elements (by passing them to procedures), or the array is intact. This works well for arrays of values, because they do not register ownership.

When sharing an array of values, the compiler can check at compile-time whether or not there is a chance of illegal sharing. However, if the sharing is legal, there is still the problem of making the array accessible to both processes. The different parallel processes do not share scope, so the *entire* array must be available to both.

One cannot simply slice the array, and give each parallel thread their own slice, because this would be ineffective, and invalidate the indexing if the different parts are not continuous. The simple solution is to give both processes the entire array, knowing that they will not share the same elements. For instance, the following code is legal, and when compiled, both assignment process classes have access to the entire `array` array:

```
PROC Main():
    INT [10] array
    SEQ:
        PAR:
            SEQ FOR i = 0 TO 4:
                array[2*i] = 1
            SEQ FOR i = 0 TO 4:
                array[2*i+1] = 2
        SEQ FOR i = 0 TO 9:
            PRINT array[i]
```

In addition to this, channels have their ownership registered, so that the scheduler will know what code to run on the other side if it needs to complete a communication. This is important, because if there is a timing requirement involving communication on a channel, the scheduler must know which code must be run before the other side is ready. As currently implemented, any process that receive a channel, also registers itself as the owner of that end of the channel. This works well with scalar channels.

It does not work well with channels of arrays. Take the example below. The two parallel processes both receives a reference to an entire array, even though they will only use disjunctive parts. If they are to register ownership, they must know which elements they and their children will use, and which they will not. However, they do not know which elements their children will use, because the indices are functions of expressions further down in the syntax tree.

```
PROC Main():
    CHAN INT [50] array
    SEQ:
        PAR:
            SEQ FOR i = 0 TO 4:
                PAR FOR j = 10*i TO 10*i + 4:
                    array[j] ! i
            SEQ FOR i = 0 TO 4:
                PAR FOR j = 10*i+5 TO 11*i-1:
                    array[j] ! i
```

The verification function in the compiler simply executes these loops and registers all elements that are used. But when the program compiles, there is only one class that represents each of the two `SEQ FOR`s, which are then instantiated multiple times. The different instances use the same source code, but because of different iterator values, they must register as owners of different channels. There are basically two ways to lift this limitation:

- Disallow expressions in indices where sharing may be violated. Simple iterator indices (like `array[i]`) in non-nested loops may be accepted, because it would then be easy to create a general function that registers channel elements.
- Evaluate the expressions that make out the different indices used, and pull then up in the syntax tree. In the above code, that means that each iteration of the first `SEQ FOR` must know that it uses array elements `10*i` through `10*i+4`, even though that expression does not appear until further down in the syntax tree. Implementing this would not be trivial, but should not be too hard, because index expressions may be limited to a few common integer operators and simple loops.

9.2.2 Minor Missing Language Features

These are features that could easily be implemented, and are a “small matter of programming”. They do not require any conceptual changes or deep thought.

Advanced Expressions

Occam has a richer expression syntax than C. There is a separate operator for addition with overflow checking and modulo (wrapping) addition, for instance. Also missing are operations on types, such as `BYTESIN`/`sizeof`. An operator for determining the number of elements in an array is also needed.

Occam contains operators for slicing arrays, and initializing them with lists. TIME/Occam may only initialize arrays to repeat a single value.

Multidimensional Arrays

Currently, only one-dimensional arrays are supported. Multidimensional arrays may be created by repeating use of the `TYPE` command, but only one dimension may be accessed at a time. Lifting this restriction is a minor change.

Allow Sending Expressions

The implementation of sending ("`!`") only allows variables as right-hand side, because the implementation just transfers the pointer. To allow sending expressions, a temporary variable must be created transparently. It must exist until the transfer is complete, and then be destructed.

Support for Hexadecimal Constants

There is currently no support for hexadecimal or octal constants. This, perhaps along with binary constants, is useful, and is relatively simple to implement.

Handling of Deadlines

The weakly hard timing constraints, and the `TIME..SKIP` mentioned in section 4.2 have not been implemented as part of the language. From a language point of view, this is a small change, although getting it to work properly with the scheduler is harder.

Defining Functions

Now, only procedures can be defined in the language, but other (external) functions can be used in expressions. Syntax for defining functions should be added to the language. Functions should be free of side-effects, which disallows any use of channels.

9.3 Missing in Compiler and Scheduler

This section contains a list of unfinished features in the implementation of the compiler and scheduler.

9.3.1 Missing in Scheduler

The scheduler is not completely implemented. The current version only does serialization, up to the point where it meets uncertain processes, misses a deadline, or finishes. That is the prediction phase. The planning and execution phases are not implemented. Also missing is a system for discovering deadlocks. Now, the scheduler hangs on a deadlock.

The scheduler also lacks the complete code for deleting processes that have finished, although some framework exists.

9.3.2 Memory Handling

The current way to handle variables and scopes has limitations. First, there is no good way of creating temporary variables without creating a new scope and delete-scope process. This makes it difficult to implement such simple things like sending a constant over a channel (`ch ! 1`).

More serious are the potential memory leaks. The memory leaks occur if a `TIME` is placed in a scope, like in the following code:

```
INT x
TIME 10:
    My_Proc();
```

The variable `x` is created even though it is outside of a timing requirement, because the scope is treated as a discovery process. `My_Proc()` is then called. However, the delete-scope process that should free memory of variable `x` is not called, because it is outside of any timing requirements. The result can be seen in figure 8.5 on page 111, where the white nodes at the bottom are mostly processes that free memory.

The delete-scope processes cannot be discovery processes. They must be run in or after the execution phase, and not at expand time (one cannot free the memory of a variable before it has been used). Freeing memory should therefore not be done by explicit processes, like in the current implementation.

9.3.3 Execution Time Estimates

The compiler writes only arbitrary execution time estimates. Incorporating correct estimates automatically is difficult, but there are two other alternatives: The first is to allow the programmer to specify the estimates, and the second is just to remember them from last time and use them again, if parameters have not changed.

The programmer can specify execution time estimates directly, by writing the times somewhere in the source code, or even provide execution time estimate functions. Say the program runs a function `my_func(int, int)`, then the programmer can provide a function `my_func_est(int ,int)`, that returns the execution time for a specific set of parameters. These estimator functions could be provided in several versions, depending on which arguments are known at expand-time (which is when the execution time estimates are needed), and which arguments that are actually relevant to the execution time.

The second alternative is to assume no knowledge of execution times when first executing code. After a piece of code is run, the compiler can then store all relevant arguments, and the execution time. When running the same piece of code the next time, the compiler can compare the arguments, and if they are equal, use the same execution time.

9.3.4 Uncertainty

The compiler does not flag the correct processes as "uncertain". As of now, all `IF`s are flagged "uncertain", but no other processes. All `IF`s, `FOR`s and `WHILE`s with conditional expressions that are not only functions of constants and iterator variables should be marked "uncertain".

The framework exists, because all variables are flagged with "knownedness" (described in section 7.3), and there exists ways to manipulate expressions with these possibly unknown variables in the compiler, in ways that preserve their status. All that remains is to use this information to mark the processes.

9.3.5 Combining and Separating Statements

Now, each statement in TIME/occam is converted into a separate node in the program state graph, with its own C++ object. There is a significant overhead involved in manipulating these nodes. An improvement would be to combine statements into fewer nodes, if they do not use communication with channels.

This would sometimes cause problems with the execution time estimates. Now, if a branch is found, it is first resolved, and then the execution time of the correct choice is added. If an entire `IF` block is to be combined into one statement, then one would have to use the worst-case estimate, which could be tricky to find.

Another flaw in the compiler is that it runs expressions in conditionals of `IF`, `WHILE`, `FOR` and more, at expand time. The execution time associated with evaluating conditionals and loop bounds is therefore not taken into account. This execution time could be arbitrarily high, because all kinds of functions are allowed in these expressions.

A solution to this problem is to put these expressions in a separate assignment process, before the `IF`, `WHILE` or `FOR`; and assign the result of all bounds and conditional expressions to temporary variables. The execution time of the expand phase of the next process would then be small. For `IF..ELSE` variants, which branch to take must be evaluated, not just the first conditional.

Chapter 10

Conclusion

Real-time programming quickly turns into a question of parallelism, as tasks often need to be run simultaneously. Using threads and pre-emptive scheduling to run parallel programs is a framework that has worked well on desktop computers. Independent threads can be transparently interleaved, blessedly ignorant of each others presence. But use it on a concurrent system, and a world of potential bugs emerges.

By definition, a concurrent system cannot be implemented as independent threads. If threads are not independent, few scheduling algorithms can be proven correct. This also applies to the popular rate-monotonic scheduling algorithm, where priorities are assigned in order of frequency. Using this algorithm on a concurrent system may quickly lead to failure, if a low and high priority thread shares a resource under mutual exclusion.

If the low priority thread has locked the resource, it may be interrupted by any other thread with a higher priority, and it may take some time before it can finish and unlock. During this time, the high priority thread will not be able to run, and, in effect, has had its priority severely decreased. To alleviate some of this, more complex schemes are needed. Using priority inheritance, some priority inversion can be avoided.

Concurrent programming can be hard enough without timing requirements; as deadlocks, starvations and race conditions may turn up in unexpected places. Using bare mutexes and semaphores makes such errors hard to trace. The concurrent mechanisms in Java improve on structure, but are also prone to errors. In Java, many mistakes can be avoided by special design patterns, but these patterns do not yield particularly clear and readable code, neither do they support the object oriented principles for which they were designed.

Perhaps the answer to these difficulties lies in structured communication and CSP? CSP-based concurrency can be checked for consistency by automated tools. A program that is more easily analyzed by a computer is likely to be easier to comprehend for human beings too. Advocates of CSP claim that it leads to fewer bugs, but the techniques have not been widely used since the disappearance of occam and the transputer.

Combining the CSP approach to concurrency with explicit formulation of timing requirements lead to the development of the TIME/occam language. In this language, timing requirements is the only driving force behind progress, and statements with no timing requirements will never be executed. Communication between threads use channels, and is synchronous. Timing requirements are also transmitted along channels. If one end has a timing requirement, then it will drive the process behind the other end.

The explicit timing requirements also allow explicit ways to deal with missed deadlines. Several different actions are possible. One possible action is to skip the task altogether. A repeated task skips one iteration. Another option is to set priorities. Priorities make more sense when applied to a system where deadlines will be missed, than to a system where deadlines are assumed to be met. If all that can be done is done in time, why prioritize? A further choice is to specify weakly hard timing requirements, for instance that out of every 30 repetitions of a task, at least 20 deadlines must be met.

Implementing such a system is a challenge. First of all, such a system demands online execution time analysis, which is notoriously difficult to do in practice. A successful implementation must be able to predict the execution and execution time of the program some time into the future. Conditional statements with complex expressions and functions that have unpredictable execution times may make this hard. Any uncertainty must be treated as a potential missed deadline. If there are too many of them, one may in practice be back to scratch, using a priority-based scheduler.

Execution time analysis is subject to much research, and good solutions exist. Execution time can be measured online and stored. Program slicing is also an interesting option, where a part of a program is run in advance, just enough to determine all branches.

In sum, TIME/occam is an interesting experiment, which provides an elegant notation for concurrent and real-time programming. However, overheads may be too high and execution time analysis too hard. These problems must be solved to make the system work in practice.

Part IV

Appendix

Appendix A

TIME/occam BNF

Non-terminals are enclosed between $\langle$ and $\rangle$. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

$$\begin{array}{lcl}
\langle \textit{Init-spec} \rangle & ::= & = \langle \textit{Exp1} \rangle \\
 & | & \epsilon
\end{array}$$

$$\begin{array}{lcl}
\langle \textit{Decl-spec} \rangle & ::= & \langle \textit{Qualifier-spec} \rangle \ \langle \textit{Type-spec} \rangle \ \langle \textit{Array-spec} \rangle \ \langle \textit{LowerIdent} \rangle
\end{array}$$

$$\begin{array}{lcl}
\langle \textit{Array-spec} \rangle & ::= & \texttt{[} \ \langle \textit{Exp1} \rangle \ \texttt{]} \\
 & | & \texttt{[ ]} \\
 & | & \epsilon
\end{array}$$

$$\begin{array}{lcl}
\langle \textit{Variable-spec} \rangle & ::= & \langle \textit{Decl-spec} \rangle \ \langle \textit{Init-spec} \rangle
\end{array}$$

$$\begin{array}{lcl}
\langle \textit{ListVariable-spec} \rangle & ::= & \epsilon \\
 & | & \langle \textit{Variable-spec} \rangle \ ; \ \langle \textit{ListVariable-spec} \rangle
\end{array}$$

$$\begin{array}{lcl}
\langle \textit{Param-spec} \rangle & ::= & \langle \textit{Decl-spec} \rangle
\end{array}$$

$$\begin{array}{lcl}
\langle \textit{ListParam-spec} \rangle & ::= & \epsilon \\
 & | & \langle \textit{Param-spec} \rangle \\
 & | & \langle \textit{Param-spec} \rangle \ , \ \langle \textit{ListParam-spec} \rangle
\end{array}$$

$$\begin{array}{lcl}
\langle \textit{Qualifier-spec} \rangle & ::= & \texttt{CONST} \\
 & | & \texttt{CHAN} \\
 & | & \texttt{IN} \\
 & | & \texttt{OUT} \\
 & | & \texttt{VAL} \\
 & | & \epsilon
\end{array}$$

```
⟨Type-spec⟩  ::=  STRING
             |    BOOL
             |    INT
             |    REAL
             |    ⟨UpperIdent⟩

⟨Program-spec⟩  ::=  ⟨ListGlobal-spec⟩

⟨Global-spec⟩  ::=  PROC ⟨CapIdent⟩ ( ⟨ListParam-spec⟩ ) :
                    ⟨SingleStBlock-spec⟩
               |    TYPE ⟨UpperIdent⟩ = ⟨Type-spec⟩ ⟨Array-spec⟩
               |    CINCLUDE ⟨String⟩
               |    CSYSINCLUDE ⟨String⟩

⟨ListGlobal-spec⟩  ::=  ε
                   |    ⟨Global-spec⟩
                   |    ⟨Global-spec⟩ ; ⟨ListGlobal-spec⟩

⟨Block-spec⟩  ::=  { ⟨ListVariable-spec⟩ ⟨ListProc-spec⟩ }

⟨SingleStBlock-spec⟩  ::=  { ⟨ListVariable-spec⟩ ⟨ListProc-spec⟩ }

⟨Order-spec⟩  ::=  SEQ
              |    PAR

⟨Else-spec⟩  ::=  ; ELSE : ⟨SingleStBlock-spec⟩
             |    ; ELSE IF ⟨Exp1⟩ : ⟨SingleStBlock-spec⟩ ⟨Else-spec⟩

⟨Proc-spec⟩  ::=  ⟨Alt-spec⟩
             |    ⟨LowerIdent⟩ ⟨Assignment-op⟩ ⟨Exp1⟩
             |    ⟨LowerIdent⟩ [ ⟨Exp1⟩ ] ⟨Assignment-op⟩ ⟨Exp1⟩
             |    ⟨CapIdent⟩ ( ⟨ListArg-spec⟩ )
             |    ⟨Comm-spec⟩
             |    DELAY ⟨Exp1⟩
             |    ⟨Exp1⟩
             |    ⟨Order-spec⟩ FOR ⟨LowerIdent⟩ = ⟨Exp1⟩ TO ⟨Exp1⟩ :
                  ⟨SingleStBlock-spec⟩
             |    IF ⟨Exp1⟩ : ⟨SingleStBlock-spec⟩ ⟨Else-spec⟩
             |    ⟨Order-spec⟩ : ⟨Block-spec⟩
             |    PRINT ⟨ListExp1⟩
             |    SKIP
             |    STOP
             |    TIME ⟨Exp1⟩ : ⟨SingleStBlock-spec⟩
             |    WHILE ⟨Exp1⟩ : ⟨SingleStBlock-spec⟩

⟨ListProc-spec⟩  ::=  ⟨Proc-spec⟩
                 |    ⟨Proc-spec⟩ ; ⟨ListProc-spec⟩

⟨Arg-spec⟩  ::=  ⟨Exp1⟩
```

```
⟨ListArg-spec⟩    ::=  ϵ
                   |    ⟨Arg-spec⟩
                   |    ⟨Arg-spec⟩ , ⟨ListArg-spec⟩

⟨Comm-spec⟩   ::=  ⟨LowerIdent⟩ ! ⟨LowerIdent⟩
               |    ⟨LowerIdent⟩ ? ⟨LowerIdent⟩

⟨Alt-spec⟩   ::=  ALT : { ⟨ListAltAlt-spec⟩ }
              |    ALT FOR ⟨LowerIdent⟩ = ⟨Exp1⟩ TO ⟨Exp1⟩ :
                   { ⟨ListAltAlt-spec⟩ }

⟨AltComm-spec⟩   ::=  ⟨LowerIdent⟩ ? ⟨LowerIdent⟩

⟨AltAlt-spec⟩   ::=  ⟨Alt-spec⟩
                 |    ⟨Exp1⟩ −> ⟨AltComm-spec⟩ : ⟨SingleStBlock-spec⟩
                 |    ⟨Exp1⟩ −> SKIP
                 |    ⟨AltComm-spec⟩ : ⟨SingleStBlock-spec⟩

⟨ListAltAlt-spec⟩   ::=  ⟨AltAlt-spec⟩
                     |    ⟨AltAlt-spec⟩ ; ⟨ListAltAlt-spec⟩

⟨Assignment-op⟩   ::=  =
                   |    *=
                   |    /=
                   |    %=
                   |    +=
                   |    −=
                   |    <<=
                   |    >>=
                   |    &=
                   |    ^=
                   |    |=

⟨ConstVal-spec⟩   ::=  ⟨Integer⟩
                   |    ⟨Double⟩
                   |    ⟨String⟩
                   |    true
                   |    false
                   |    ⟨Char⟩

⟨Exp1⟩   ::=  ⟨Exp1⟩ || ⟨Exp2⟩
          |    ⟨Exp2⟩

⟨Exp2⟩   ::=  ⟨Exp2⟩ && ⟨Exp3⟩
          |    ⟨Exp3⟩

⟨Exp3⟩   ::=  ⟨Exp3⟩ == ⟨Exp4⟩
          |    ⟨Exp3⟩ != ⟨Exp4⟩
          |    ⟨Exp4⟩
```

```
⟨Exp4⟩       ::=  ⟨Exp4⟩ < ⟨Exp5⟩
               |  ⟨Exp4⟩ > ⟨Exp5⟩
               |  ⟨Exp4⟩ <= ⟨Exp5⟩
               |  ⟨Exp4⟩ >= ⟨Exp5⟩
               |  ⟨Exp5⟩
⟨Exp5⟩       ::=  ⟨Exp5⟩ | ⟨Exp6⟩
               |  ⟨Exp6⟩
⟨Exp6⟩       ::=  ⟨Exp6⟩ ^ ⟨Exp7⟩
               |  ⟨Exp7⟩
⟨Exp7⟩       ::=  ⟨Exp7⟩ & ⟨Exp8⟩
               |  ⟨Exp8⟩
⟨Exp8⟩       ::=  ⟨Exp8⟩ << ⟨Exp9⟩
               |  ⟨Exp8⟩ >> ⟨Exp9⟩
               |  ⟨Exp9⟩
⟨Exp9⟩       ::=  ⟨Exp9⟩ + ⟨Exp10⟩
               |  ⟨Exp9⟩ − ⟨Exp10⟩
               |  ⟨Exp10⟩
⟨Exp10⟩      ::=  ⟨Exp10⟩ * ⟨Exp11⟩
               |  ⟨Exp10⟩ / ⟨Exp11⟩
               |  ⟨Exp10⟩ % ⟨Exp11⟩
               |  ⟨Exp11⟩
⟨Exp11⟩      ::=  − ⟨Exp11⟩
               |  + ⟨Exp11⟩
               |  ⟨Exp12⟩
⟨Exp12⟩      ::=  ⟨LowerIdent⟩ [ ⟨Exp1⟩ ]
               |  ⟨LowerIdent⟩ ( ⟨ListExp1⟩ )
               |  ⟨Exp13⟩
⟨Exp13⟩      ::=  ⟨ConstVal-spec⟩
               |  ⟨LowerIdent⟩
               |  ( ⟨Exp⟩ )

⟨Exp⟩  ::=  ⟨Exp1⟩

⟨ListExp1⟩   ::=  ϵ
               |  ⟨Exp1⟩
               |  ⟨Exp1⟩ , ⟨ListExp1⟩
```

Bibliography

[1] Dr. Nimal Nissanke: *Realtime Systems*, Prentice Hall 1997.

[2] G. Bernat, A. Burns, Albert Llamosí: *Weakly Hard Real-Time Systems*, University of York/Universitat de les Illes balears, 1 September 1999.

[3] C.L. Liu and L. W. Layland: *Scheduling Algorithms for multiprogramming in a hard real-time environment*, Journal of the ACM, vol. 20, no. 1. 1973.

[4] J. Lehoczky, L. Sha, Y. Ding: *The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behaviour*, Department of Statistics/Department of Computer Science Carnegie Mellon University. 1989.

[5] M. Törngren: *Fundamentals of implementing real-time control applications in distributed computer systems*, Dept. of Machine Design, Royal Inst. of Technology, Stockholm. 1998.

[6] Alan Burns, Andy Wellings: *Real-Time Systems and Programming Languages*, 3rd Edition, Pearson, 2001.

[7] Nancy Leveson, University of Washington: *Software: System Safety and Computers*, Addison Wesley, 1995

[8] Wikipedia: *Northeast Blackout of 2003* (read June 6th 2007).

[9] Microsoft Research: *Priority Inversion Historical Notes*, http://research.microsoft.com/ mbj/Mars_Pathfinder. (read July 26th, 2007).

[10] Victor Yodaiken: *Against Priority Inheritance*, Finite State Machine Labs. July 9th 2002.

[11] D. Locke *Priority Inheritance: The Real Story.* LinuxDevices.com, `http://www.linuxdevices.com/articles/AT5698775833.html`. July 16th 2002.

[12] Jeff Magee, Jeff Kramer: *Concurrency, State Models and Java Programming*, 2nd Edition, Wiley, 2006.

[13] Steve Schneider: *Concurrent and Real-time Systems: The CSP Approach*, Wiley, 2000.

[14] Øyvind Teig: *Safer multitasking with communicating sequential processes*, Embedded Systems Europe, June 2000.

[15] Inki Hong, Darko Kirovski, Gang Qu, Miodrag Potkonjak, Mani B. Srivastava: *Power Optimization of Variable-Voltage Core-Based Systems*, IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, Vol 18, No 12, December 1999.

[16] Bo Zhai, David Blauuw, Dennis Sylvester, Krisztian Flautner: *The Limit of Dynamic Voltage Scaling and Insomniac Dynamic Voltage Scaling*. IEEE, November 2005

[17] David Snowdon, Sergio Ruocco and Gernot Heiser: *Power Management and Dynamic Voltage Scaling: Myths and Facts*, National ICT Australia and School of Computer Science and Engineering, University of NSW. September 16th, 2005.

[18] John L. Hennessy and David A. Patterson: Computer Architecture: A Quantitative Approach, 3rd/4th edition, *Morgan Kaufmann*, 2006.

[19] Martin Korsgaard: *A Study of Worst-Case Execution Time Analysis on Complex Hardware, with Focus on Cache*, Institute of Engineering Cybernetics, Norwegian University of Science and Technology. March 2007.

[20] Raymund Kirner, Peter Puschner: *Classification of WCET Analysis Techniques* Institut für Technische Informatik, Technische Universität Wien

[21] Ronald L. Graham: *Bounds on Multiprocessing Timing Anomalies*, SIAM Journal of Applied Mathematics

[22] Thomas Lundqvist: *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories.* PhD Thesis, Department of COmputer Engineering, Chalmers University of Technology, 2002

[23] Jakob Engblom: *Analysis of the Execution Time Unpredictability cased by Dynamic Branch Prediction.* Dept. of Information Technology, Uppsala University, RTAS'03

[24] Kernighan, Brian W., Dennis M. Ritchie: *The C Programming Language, 2nd ed.*, March 1988, Englewood Cliffs, Prentice Hall.

[25] SGS-Thompson Microelectronics LTD: *occam 2.1 reference manual*, May 12, 1995, Prentice Hall

[26] Markus Forsberg, Aarne Ranta: *The Labelled BNF Grammar Formalism*, Department of Computing Science, Chalmers University of Technology and the University of Gothenburg, 2005

[27] Emden Gansner, Elefherios Koutsofios and Stephen North: *Drawing graphs with* dot. Can be found on `www.graphviz.org`. January 26th, 2006.

[28] Michael Pellauer, Markus Forsberg, Aarne Ranta: *BNF Converter: Multilingual Front-End Generation from Labelled BNF Grammars. Technical Report.*, Department of Computing Science, Chalmers University of Technology and the University of Gothenburg, 2004

[29] Charles Donnelly and Richard Stallman: *Bison, The Yacc-Compatible Parser Generator*, 30 May 2006, Bison Version 2.3