2009:131 CIV

MASTER'S THESIS

Developing for Concurrency

- A Graphical Framework for Development of Concurrent Programs

Andreas Bofjäll

Luleå University of Technology

MSc Programmes in Engineering
Space Engineering
Department of Computer Science and Electrical Engineering
Division of EISLAB

2009:131 CIV - ISSN: 1402-1617 - ISRN: LTU-EX--09/131--SE

Developing for Concurrency: A Graphical Framework for Development of Concurrent Programs

Andreas Bofjäll

Luleå University of Technology
Dept. of Computer Science and Electrical Engineering
EISLAB

andreas@bofjall.se

Sep 8th, 2009

ABSTRACT

The report describes the development of a concurrency model suited for real-time systems and a graphical language for describing the concurrency aspects of a system in a simple and intuitive way. Further, a graphical system design tool was developed, intended to help a programmer design a system which generates a code skeleton in the Timber programming language from a graphical design. To maximize platform indepence of the tool, it was developed in Java.

PREFACE

The work presented in this report is my M.Sc. thesis. It was done during the later part of 2008 as well as the spring of 2009 at the Embedded Internet Systems Laboratory (EISLAB), Department of Computer Science and Electrical Engineering at Luleå University of Technology.

Among the people I would like to thank is my supervisor Per Lindgren as well as Jonas Jonsson, Andreas Nordin and Sebastian Själin, whom I shared my office with from November to May. Your help and support has been greatly appreciated.

Last, but absolutely not least, I would like to very specially thank Andrey Kruglyak for his invaluable expertise and the countless hours spent with my work and report.

Andreas Bofjäll

Contents

Chapter 1
Introduction

1.1 The Timber programming language

A real-time system usually operates under different constraints than a desktop application; the most important ones are probably timing constraints. To simplify the development of such systems, a programming language for event-driven systems where timing constraints are built-in into the language itself, called *Timber* (short for TI*me* eMBE*dded* R*eactive*) has been developed. The principal developer of the language has been Johan Nordlander at Luleå University of Technology.

The Timber language is a *reactive, object-oriented functional language* with *object-level concurrency* and *static type checking.*

There are currently no design tools for the Timber programming language and the language may have a high learning threshold for new programmers if they are not used to functional languages or to the reactive style of programming.

Typically, the first phase of real-time system development is to create a general program structure, which implies making decisions on the concurrent aspects of the system. The concurrent behavior of the program is often the key to ensuring responsiveness of the system.

The proposed framework should enable the programmer to jump-start the development by generating a code skeleton from the system design.

1.2 Aim

The aim of this thesis is to develop a concurrency model suited for real-time systems, and a graphical language for describing the concurrency aspects in a simple and intuitive way. Furthermore, a graphical design tool intended to help a programmer with the design of a system in terms of concurrency is to be developed. A backend to generate a Timber code skeleton from the design should also be implemented.

The concurrency model should be designed based on a survey of existing approaches to both expressing concurrency and implementing it. The tool should meet the following requirements:

- The concurrency model should be simple and intuitive.
- The user interface should be user-friendly and have a low learning threshold.
- The backend should be (easily) replaceable, e.g. replaced by one producing a code skeleton in another language than Timber.
- The tool should be developed in the Java programming language to maximize platform independence.

1.3 Scope

In this work, only a backend for the Timber language has been developed, while it is reasonable to assume that the concurrency model is general enough to allow system design using other programming languages as well. Moreover, the concurrency model and design tool are aimed at a specific class of embedded, real-time systems and as of today can only express a non-hierarchical, static structure with no support for componentization.

CHAPTER 2
Background

2.1 Parallelism and concurrency

Parallel computing means executing multiple tasks at the same time. This is achieved by *multiprocessing* where each task is executed on its own processor or processor core [Schneider & Andrews, 1986].

In general, this can be very beneficial to real-time systems. The responsiveness of the system may be greatly increased since the system does not need to complete the present computation to process a new event since it may simply start a new task on a different processor to handle the incoming event.

If a computation can be *parallelized*, divided into multiple parts that can be run in parallel, increasing the number of processors can be a cost-effective way of improving the performance of the system.

While running tasks in parallel may be beneficial for shortening the response time of the system, it is offset by the need for multiple processors. If a task is seldom invoked, or completes very quickly, the processor will idle most of the time.

A concept closely related to parallelism is *concurrency* [Schneider & Andrews, 1986]. If two tasks *foo* and *bar* are concurrent, the order in which they execute is not predetermined. They can be executed either

- *foo* and then *bar*,
- *bar* and then *foo*,

- *foo* and *bar* at the same time (in parallel),
- or alternating between *foo* and *bar*.

Notice how the fourth alternative can be viewed as an emulation of true parallelism. The major difference is that in this case two tasks share a common processor. Since two tasks running concurrently may either share a single processor or execute at the same time on multiple processors, concurrency is a more general concept than parallelism. How the tasks are actually executed will depend on the *concurrency model* implemented in the programming language and/or the run-time system.

If the scheduler makes few context switches, or if few tasks are executed at the same time, the model has a *lower degree of parallelism.* This has some positive effects, such as minimizing the scheduling overhead. A lower degree of parallelism can also make problems with shared data easier to handle.

A higher degree of parallelism, however, may increase the ability of the system to process several reactions at the same time, thus shortening response times and potentially improving throughput.

When multiple tasks share a single processor, their execution must somehow be interleaved one at a time. This interleaving is accomplished by having a single task, called the *kernel*, assign the processor to each process in turn. All other tasks are temporarily suspended.

If this interleaving is done fast enough, it may seem as if several tasks are running in parallel on a single processor, when in fact they are regularly *preempted* by the kernel [Berry, 1993].

The kernel itself could be invoked at regular time intervals by an external timer interrupt or by some other method.

2.1.1 Determinism and different kinds of parallelism

A *deterministic* system always produces the same output for the same input and initial system state. If, on the other hand, the output may be different for the same input and system state, either in value or in the ordering of the values produced, the system is *non-deterministic.*

The computation of a deterministic system can often be parallelized. Consider the simple case in figure 2.1 of a program consisiting of three statements. Each of these statements is *independent* of the other two; their outcome does not affect the result of the each other and only one of them produces an *observable* effect, the printing of z. The three statements may be executed in any order and indeed parallelized without affecting the result.

Another example of a deterministic program is the `map()` function in languages such as Haskell which applies a specified function to each element in

cycle	*statements*
0	`x = 0`
1	`y = a`
2	`print z`

Figure 2.1: A simple program where the individual statements may be executed in any order.

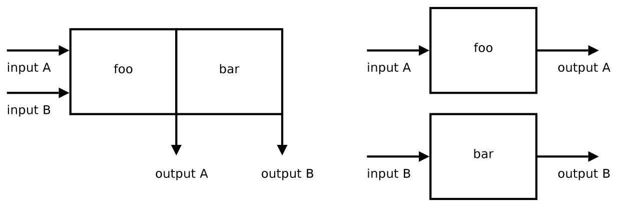


Figure 2.2: Two independent tasks can be parallelised by allowing non-determinism.

a list. Clearly, this can be done in any order and indeed parallelized. Parallelism that can be introduced without affecting the result of a computation is called *functional parallelism* and is typically not part of a concurrency model.

If, however, introducing parallelism into a system also introduces non-determinism, it may affect the correctness of the system. It may therefore be tempting to disallow this kind of parallelisation; however, allowing it often results in higher responsiveness and possibly higher system throughput. The concurrency model must therefore allow us to express the correct system behavior.

Consider the example system in figure 2.2 with two tasks, each independent of the other task. If this system is run in sequence with *foo* before *bar*, output *A* will always be produced before output *B*. If, on the other hand, the system is parallelized, one cannot know which output will be produced first. Such a system is non-deterministic although, if the ordering of the outputs does not matter, it still displays correct behavior.

When two tasks share one or more resources, such as a state variable, not only the ordering of the outputs but the *values* may be affected by the order in which the tasks are executed. This in another case of non-determinism that may affect the correctness of the system.

2.2 Concurrent models of computation

A *concurrent model of computation*, or *concurrency model*, describes how the code should be interpreted in terms of parallel and concurrent execution at run-time. It provides an abstraction that allows the programmer to express what parts of the program should be executed concurrently or in sequence and at what points independent parts of the program should be synchronized.

One example is the model based on the abstraction of *threads*, for example as implemented by the POSIX threading library. Here, the programmer explicitly defines when to spawn new threads for concurrent execution as well as when to synchronize them using mutexes, semaphores, thread joining mechanisms and the like.

Another example is *object level concurrency* where every object is by default executed concurrently with all others. Here, the programmer does not concern him- or herself with explicitly defining when to spawn new tasks of concurrent execution; they are *implicitly* defined from the object structure of the program. The programmer must, however, still explicitly define when to synchronize the objects using some synchronization construct.

The choice of concurrency model is often limited by the programming language chosen; many languages, such as Java, Ada and Timber, have their own concurrency model defined in the language specification, although this definition is often implicit. Others, such as C, can use any concurrency model provided by an external library such as the POSIX threading library.

2.2.1 Declarative and imperative concurrency

Lee & Neuendorffer [2005] divide the concurrency models into two major groups: *declarative concurrency* and *imperative concurrency.*

In imperative concurrency, a programmer defines a number of tasks that capture the desired functionality of the system. Each task executes whenever called upon by the environment. Whenever input to the task, either from the environment or another task, is ready to be processed, an instance of the task is created and supplied with the data as a parameter. This may, depending on the model, enable multiple instances of a single task to execute concurrently if data is supplied faster than it can be processed.

This contrasts with declarative concurrency where the tasks may be viewed as executing in infinite loops, continually waiting for data. The first statement is often a blocking read call; when data is supplied, the task processes the data and then returns to the blocking read call. Each task is one of a kind; they are not created in response to input data but exist independently. If data is supplied faster than the task can process it, the data can either be queued or ignored; no new task may be spawned to process it. The

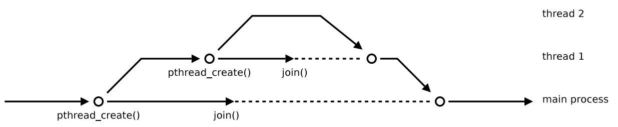


Figure 2.3: A system with two POSIX threads; after a while, the main process synchronizes with both threads using the *join* function.

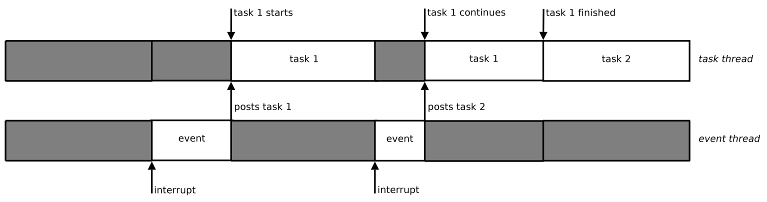


Figure 2.4: An example of a TinyOS program.

programmer declares the *flow of data* between the tasks.

POSIX threads

A classic example of imperative concurrency is the concurrency model implemented by the POSIX threading library. The library provides functions to create and manipulate threads. Synchronization is done using the `join()` function and mutexes.

The threads are not statically defined for a particular program; the programmer is free to spawn a new thread whenever he or she feels like it using the `pthread_create()` function.

Figure 2.3 displays a system with two threads; one spawned by the main process and one by the spawned thread. The program is then synchronized using the `join()` function.

TinyOS

One example of a system using an imperative concurrency model is *tinyOS* [Levis, 2006]. Here, two types of components can be declared: event handlers and tasks. A task will only execute whenever the processor is idle and event handlers have priority.

Event handlers are reactive components that execute as a response to an interrupt. An event handler cannot be preempted and therefore should either execute very quickly or start a separate task to do processor-intensive work.

A task only executes whenever the CPU is not serving an event handler and only one task may execute at a time[1]. The scheduling is illustrated in figure 2.4 where there are two event handlers and two tasks. The first event handler starts *task1* which is not finished by the time the second event handler starts which tries to start *task2*. *task2* ends up in the task queue and has to wait for *task1* to finish.

If not properly designed, one task may use the entire processor all the time, starving other tasks.

TinyOS does not have mutexes but has an `atomic`-keyword which guarantees that a block of code is not interrupted by an event handler.

Process networks

An example of a system using a declarative concurrency model is a *Kahn process network* [Stefanov et al., 2004]. Each component is implemented by a separate, infinitely running thread. A thread uses blocking read calls to get data and posts the data to its output whenever processing is complete. No check is performed whether or not the receiver is ready for the output and the operating system queues the data if necessary.

This model has several good characteristics, notably being deterministic and the blocking read call can be easily and efficiently realized in both hardware and software.

Synchronous languages

Another class of languages using a declarative concurrency model are the *synchronous languages*. A synchronous language executes in discrete time steps. All components execute simultaneously in every time step, but a component can sometimes be configured to run only on every nth tick. This simplifies synchronization and locking since it is known beforehand when a component will execute.

An example of a program defined using a synchronous language is displayed in figure 2.5. The program has three components: *foo* executes on every clock tick, and *bar* and *quz* executing on every second tick. They are, however, offset one tick from each other and may then share a resource without locking since their execution will never be interleaved.

The *Lustre* language used in the SCADE[2] development environment is an example of such a language; SCADE was created for safety-critical embedded systems and used in avionics [Lee & Zhao, 2007, Scade manual, 2007].

Another example is the *Esterel*[3] language, designed for the development of complex, reactive real-time systems.

[1] A multi processor version does not seem to be available.
[2] `http://www.esterel-technologies.com/products/scade-suite/`
[3] `http://www-sop.inria.fr/meije/esterel/esterel-eng.html`

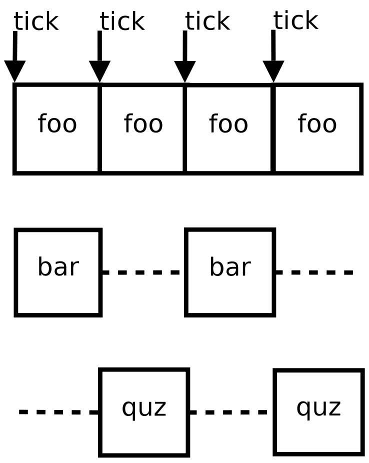


Figure 2.5: A synchronous language with three components.

Ada concurrency model

The *Ada* programming language was developed in the late 1970s and 1980s by a commission of the United States Department of Defense to supersede the then hundreds of languages in use by the department. Ada is known for its readable (yet somewhat verbose) code and run-time safety checks. Many Ada compilers have been validated for use in critical applications such as in avionics and space; for instance, Ada was used to develop the fly-by-wire systems in the Boeing 777 airplane. The latest standard is from 2005 [Ada manual, 2006].

Originally targeted at critical, real-time, embedded systems, the Ada language includes numerous run-time checks against problems such as unallocated memory and buffer overflows as well as static type checking.

The concurrency model is based on *tasks*, much like threads. Each task can communicate by passing messages and sharing variables with other tasks. When instantiated, a task waits for input, processes it and exits. It can be implemented as an infinite loop, enabling a task to wait for input, process it and then wait for input again. A task can also be started periodically by blocking on a delay statement. Since the tasks are statically defined and are active all the time, we say that the Ada language uses a declarative concurrency model.

Synchronization between tasks is accomplished in Ada by using mutexes and semaphores as well as *protected objects*, which were introduced in the Ada 95 standard [AdaCore, 2009]. A protected object is an object executing under mutual exclusion.

Giotto concurrency model

The *Giotto* toolbox provides an abstract model directed at the implementation of control systems in real-time, embedded systems [Henziger et al., 2001]. The model separates the platform dependent parts from the platform independent parts and tries to offer more flexibility in choosing an embedded platform as well as to aid the automated validation and synthesis of code.

Giotto tries to bridge the gap between control engineers designing mathematical algorithms governing a control system and software engineers implementing them in executable code. Henziger et al. [2001] divides the design process into three steps, with Giotto covering the second part:

1. Process modelling and derivation of control laws by a control engineer,
2. functionality and timing of periodic software tasks and mode switches, and
3. hardware mapping and computation by a software engineer.

The basic concurrency unit in Giotto is a *task*. One or more tasks are grouped together into a *mode*. Tasks communicate by calling *drivers*; a driver should be a small amount of code, executed almost instantly on the target platform. A driver cannot be preempted by a change in the environment and is seen as an atomic transaction.

Giotto itself does not make any assumptions on the *implementation* of the concurrency model; this can be done with multitasking on a single processor as well as multiprocessing on more than one, or with any combination of these approaches. The Giotto model only guarantees functionality and timing of the program [Henziger et al., 2001].

The compiler can be aided by the programmer by giving *platform constraints* that may, for example, assign a task to a specific processor or assign a specific priority to a task.

All communication between components in Giotto is done through *ports*. A port can be written to by the environment or a task; when read from, it returns the latest value written to it. Ports can be internal to a task (state variables), written to by the environment (*sensor ports*), output ports written to by the program (*actuator ports*) and ports for communicating between tasks (*task ports*).

Each task is periodic and has a set of input ports and output ports. An assigned driver provides the data to the input ports. The driver can be *guarded*; invoking the task only if a certain condition evaluates to true.

The internal task function is implemented as a sequential program, written in an arbitrary language, translating input data to output. The task function cannot be terminated prematurely and can not synchronize internally with other tasks; all synchronization is explicitly defined outside the task itself. The model therefore has no need for mutexes and the like. However, the Giotto compiler needs to know the worst case execution time after which the function needs to have provided its output values to the output ports.

Each mode executes a fixed collection of tasks with a prespecified frequency. A task may belong to more than one mode. Tasks in different modes may share input and output ports. When changing modes, one task may be swapped with another.

Changing mode is called *mode switching*. Values can be passed between modes by using *mode ports*.

Giotto targets control systems in avionics and other critical systems and this is reflected in its design. It does not support aperiodic or sporadic tasks and event-driven designs are not possible. This simplifies analysis considerably at the cost of flexibility.

Since the concurrency model in Giotto is statically defined at compile-time without allowing the programmer to spawn new tasks, we can say that Giotto uses a declarative concurrency model.

2.3 Explicit and implicit concurrency

Awad & Ziegler [1997] further split concurrent programming into *explicit* and *implicit* concurrency, writing:

> In the implicit model, the objects themselves have concurrent execution capabilities whereas in the explicit model, objects are encapsulated inside processes, the latter providing concurrent execution capabilities.

2.3.1 Explicit concurrency

When using a concurrency model with explicit concurrency, the programmer specifically states when to spawn a separate thread of execution.

One example of explicit concurrency is programming using POSIX threads in C or when using the threading capabilities of a programming language such as Java.

Threads, however, are not really suited to real-time embedded systems. They present severe problems in terms of data synchronization, deadlocks and priority inversion [Lee, 2006, Sanchez et al., 2006].

The explicit model relies heavily on the programmer making the concurrency decisions; this may be beneficial if the programmer is able to make better decisions than the compiler. While the programmer may know more

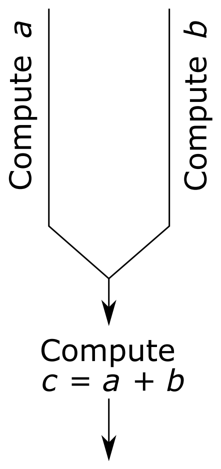


Figure 2.6: A program where a computation depends on the result of two previous computations.

about the system as a whole, optimizing the program for an explicit concurrency model is more demanding for the programmer.

2.3.2 Implicit concurrency

The basic assumption in implicit concurrency models is that the building blocks of a system, be they tasks, objects, or something else, can execute concurrently and the programmer explicitly defines when to synchronize them.

The responsibility for making concurrency-related decisions is moved from the programmer to the compiler and the run-time system in an implicit concurrency model. While this may be less demanding for the programmer, it requires well-written compiler and run-time system. Moving responsibility from the programmer may also help mitigate concurrency-related problems, such as deadlocks, if the programmer is not an expert in concurrent programming.

2.4 Shared resources and synchronization

One of the major difficulties with concurrency programming arise from shared resources and synchronization [Burns & Wellings, 2001]. The problem of *synchronization* of concurrent threads of execution arises when a certain order of operations is required. Figure 2.6 displays an example of a program where a computation depends on the result of two previous computations. The two parts of the program that may execute concurrently must be synchronized to ensure that c is calculated with the correct values of a and b.

cycle	task *foo*	task *bar*	task *foo*	task *bar*
0	`t = x`		`t = x`	
1	`x = t + 1`			`s = x`
2		`s = x`	`x = t + 1`	
3		`x = s + 1`		`x = s + 1`

Figure 2.7: Two possible ways of executing the same two tasks with different results; the result is non-deterministic.

cycle	thread *foo*	thread *bar*
0	`lock m`	
1	`t = x`	`lock m`
2	`x = t + 1`	...
3	`unlock m`	...
4		`s = x`
5		`x = s + 1`
6		`unlock m`

Figure 2.8: Solving the mutual access problem with a mutex.

Synchronization is also a way to guarantee state consistency by ensuring exclusive access to a resource. Consider the following situation: two tasks share a variable x. The variable x is used to count the number of invocations of both tasks, they therefore increase it every time they are invoked. This involves a three-step process: first, x is read, then increased by one and, finally, x is updated with the new value.

If the two tasks would try to update x at the same time, the update algorithm might not produce the intended result. The problem is illustrated in figure 2.7 where the two tasks are interleaved in two different ways, producing two different values of x. This is another example of non-determinism[4], in this case the output depends on both the original value of x and how the tasks are interleaved.

If we want to guarantee a certain system behavior, this problem can be solved in many ways, for example, by using *mutexes*. A mutex is an exclusive lock that can only be owned by one task at a time. By careful programming, a program can use mutexes to only allow one task at a time to access a shared resource. Solving the problem with a mutex is displayed in figure 2.8.

Note, however, that using mutexes is prone to such problems as deadlock and priority inversion [Lee, 2006, Sanchez et al., 2006].

[4]See section 2.1.1 on page 4.

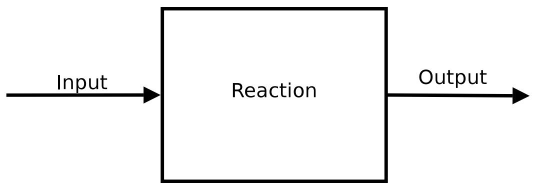


Figure 2.9: A reaction.

2.5 Reactivity

Another important aspect of a real-time system is the notion of *reactivity*. In a reactive system, the system reacts to changes in the environment. If the environment is static, the system is idle. This can be contrasted with a non-reactive system where the system continually checks if any change has occured and then invokes a method if a change has been detected. A reactive system often has a higher responsiveness to changes in the environment [Nordlander et al., 2002].

2.5.1 Reactions and reaction chains

The behavior of a reactive system is defined in terms of one or more *reactions*, each having one input and possibly one or more outputs. They are to be considered as discrete events, the input being triggered by the environment and the output produced by the system when it has finished processing. A reaction is illustrated in figure 2.9.

A reaction may be divided into parts forming what is known as *reaction chains*. Each of these parts may be viewed as a separate reaction with one input being the output of the part before it. However, a part may also trigger another part *concurrently*, enabling the original reaction to have more than one unsynchronized output.

2.6 Timing behavior

In a real-time system, the *timing* of a reaction is just as important as its correctness. Every reaction has timing requirements that need to be defined. Often used concepts are the *baseline* and the *deadline* for a reaction.

A real-time system may need to ensure that a reaction is completed within a certain time window, called the *deadline* [Burns & Wellings, 2001].

Nordlander et al. [2002] also suggests defining a *baseline* for each reaction. The baseline is defined as the system time when a reaction is to be invoked, all deadlines within the reaction are then measured relative to its baseline.

This allows for defining delayed reactions, where the new baseline is offset by a certain amount of time which allows the definition of periodic processes in terms of reactions.

The time between the baseline and the deadline when the method is allowed to execute is called the *permissible execution window* [Lindgren et al., 2008]. If execution continues past the deadline, the method has missed its deadline which can be considered fatal in a hard real-time system.

2.7 Timber concurrency model

In the Timber programming language, the concurrency is implemented at the object level [Nordlander et al., 2002], so every object can be executed concurrently with all others. However, only one method at a time is allowed to execute in each object to protect the consistency of its internal state. No further locking, such as mutexes and the like, is available, neither is it necessary. Resources such as state variables are encapsulated in objects and have to be accessed via the object's methods.

In Timber, system functionality is defined in terms of reactions, each one involving one or several objects. To guarantee liveness, no blocking calls to the environment or infinite loops are allowed [Lindgren et al., 2005].

The programmer explicitly defines all interactions between the objects. The programmer is free to create multiple objects of the same class to process multiple (identical) input events at runtime. Timber is therefore based on an imperative concurrency model; had it been declarative, only one instance of each class would be allowed and it would be created when the program is first started.

Timber uses object-level concurrency, thus all objects may execute concurrently if required by the environment that triggered the reaction. This matches an implicit concurrency model where all the basic building blocks of a system execute concurrently. Had Timber used an explicit concurrency model, the programmer would need to explicitly state when to spawn a new task; this is not applicable to a Timber program.

Every reaction has a baseline and deadline, in line with the discussion above. Timber also permits *delayed reactions*, a reaction that is to be invoked at a certain time in the future, which can be used (among other things) to encode periodical tasks.

2.7.1 An example system

An example system with three objects is displayed in figure 2.10. First, method *foometh1* is invoked in object *foo* by some external event. This method invokes *barmeth1* in object *bar* and *quzmeth1* in object *quz*, both of

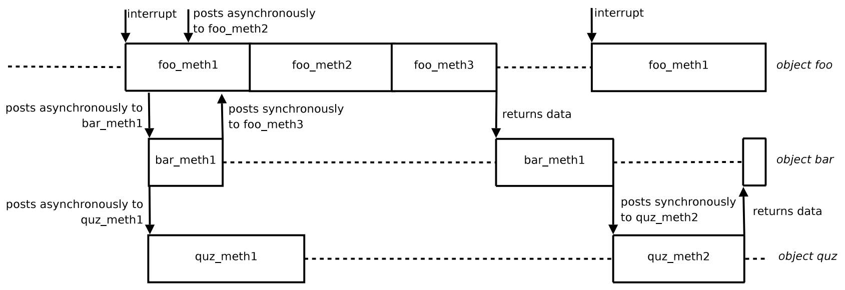

interrupt
posts asynchronously to foo_meth2
interrupt
foo_meth1
foo_meth2
foo_meth3
foo_meth1
object foo
posts asynchronously to bar_meth1
posts synchronously to foo_meth3
returns data
bar_meth1
bar_meth1
object bar
posts asynchronously to quz_meth1
posts synchronously to quz_meth2
returns data
quz_meth1
quz_meth2
object quz

Figure 2.10: An example of a Timber program.

which are executed concurrently.

A series of method calls then follows; *foometh1* calls *foometh2* to be executed concurrently. Since they are in the same object, execution is delayed until the method has finished. *barmeth1* calls *foometh3* to retrieve data; *barmeth1* therefore waits until *foometh3* has executed and the data is returned. *barmeth1* then calls *quzmeth2* in the same way and needs to wait until the data is returned.

quzmeth1 does not make any calls and finishes execution before *quzmeth2* is invoked later.

CHAPTER 3
Concurrency model

3.1 Reactive model

Our concurrency model will target real-time systems. This affects a number of properties, most notably the need to maximize responsiveness and meet timing constraints.

A reactive system often has a higher responsiveness to changes in the environment than a non-reactive model while allowing for a better use of resources [Nordlander et al., 2002]. A model should reflect fundamental properties of the system, hence the concurrency model should also be reactive.

Design choice 1. The concurrency model will be reactive; the functionality of the system will be defined as a number of reactions to discrete external events.

In order to encode timing constraints and enable the programmer to verify them, each reaction will have a baseline [Lindgren et al., 2005] and a deadline [Burns & Wellings, 2001], as discussed in 2.6.

Design choice 2. Each reaction will have a baseline and a deadline defined. The time inbetween is called the *permissible execution window* for the reaction.

In addition, we will introduce the notion of *objects* into our model. An object is a part of the system and contains *methods* and *state variables.* A number of methods, which may reside in different objects, taken together form a reaction to an external event.

In order to increase responsiveness, the concurrency model should maximize achievable parallelism. Nordlander et al. [2002] recommends to use objects as units of concurrent execution and this is also the approach used in Timber [Lindgren et al., 2005] (Timber also requires that at most one method may execute in each object at a time to ensure state consistency).

Design choice 3. We adopt an object-level concurrency model. In this model, each object may execute concurrently with all other objects. However, only one method in each object will be allowed to execute at any single time to maintain state consistency.

3.2 Sequential execution blocks

When using object-level concurrency, dividing the system into objects becomes a delicate task since the design in terms of objects will also affect responsiveness and throughput. Delaying this division into objects to a later stage in the system design can be beneficial.

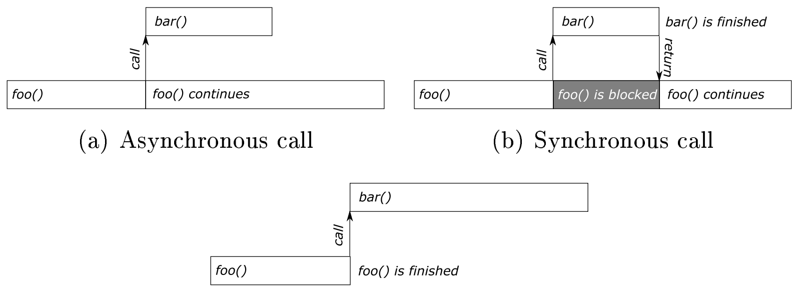
The system will therefore be designed using the smallest component possible: a block of *sequential code* (which may be smaller than a method in an object). Functional parallelism is, however, allowed to be introduced for sequential code by the compiler, as it does not affect the determinism of system behavior, but this is outside the scope of our concurrency model.

A *reaction* will consist of one or more of these *sequential execution blocks* or *code blocks*. In keeping with the notion of reactivity, the first code block in a reaction is triggered by an input event. A *reaction chain* will consist of one or more reactions and synchronous and asynchronous calls between them. Therefore, parts of the same reaction chain (but not parts of the same reaction) may be executed concurrently as discussed in the next section.

3.2.1 Interactions between code blocks

Each code block may communicate with other code blocks. A distinction is made between three types of calls: an *asynchronous call*, a *synchronous call* and a *sequential call* as in figure 3.1.

A sequential call is a call that permanently hands over execution to the target code. The caller ceases execution and the target code takes over and does not return. Since a sequential call permanently hands over execution and never returns, a sequential call may only be performed when a code block has finished executing. This may include a sequential call to the code block *itself*, enabling the possibility of loop constructs, see section 4.2 for an



(a) Asynchronous call (b) Synchronous call

(c) Sequential call

Figure 3.1: Asynchronous, synchronous and sequential calls.

example. A sequential call is illustrated in figure 3.1(c).

A code block called asynchronously may be executed concurrently with the calling code block. If, however, it is called synchronously, the caller will wait for the called method to finish, optionally returning a result. This is illustrated in figure 3.1(b).

3.2.2 State variables

Sequential code blocks may have an internal state preserved between invocations and they thus need to be *stateful*.[1]

Design choice 4. A sequential code block may be stateful.

To keep the concurrency model simple and intuitive, a state variable is always introduced in the global namespace. It may be freely used along a reaction but, to ensure state consistency, if two reactions share a state variable the second reaction cannot start until the first one has finished executing. If this is not the intended behavior, access to the variable needs to be explicitly synchronized by the programmer. Such synchronization can be achieved by encapsulating the variable in a shared code block, which is then called synchronously from both reactions. This solution preserves a higher degree of concurrency in the system.

Note that only one instance of a code block can execute at any single time since multiple instances of a single code block running concurrently would share resources such as state variables and they may interfere with each other.

[1]If not, the output of the system could only depend on the present input and not any previous state. Such a system would not be very useful in real life.

3.3 Declarative, implicit and timing

The choice between declarative and imperative concurrency is perhaps not an easy one. Many factors need to be taken into account before any model is chosen.

However, a real-time system does have special constraints. If the model is applied to an embedded system, even further restrictions apply. Such a system needs to be able to respond quickly to changes in the environment (such as sensor readings or user inputs), it may need to use as little power as possible and, for safety-critical real-time systems, ensure complete correctness including timely behavior.

In a declarative model, data is simply handed to a task, in the hope that the task is ready to process it. This may make it more difficult to analyze the timing of the system to ensure correct timing behaviour.

Another drawback of declarative models is that a system where a single task should be triggered whenever any one of multiple inputs is ready is quite complex. It has to be implemented either by constantly checking for new inputs in a loop or by splitting the task into multiple subtasks, each blocking on one input.

On the other hand, in an imperative model, it is easier to design a system with one task per input and any number of tasks may be spawned to process multiple events.

The real world can be seen as reactive and event-based and it can even be argued that the reactive approach is more intuitive to the programmer [Nordlander et al., 2002].

Design choice 5. The concurrency model will be imperative.

Next, we need to choose between using an explicit or implicit concurrency model. Design choice 3 stipulates concurrency on the object level; however, since the division into objects is to be made at a later stage, the object structure of the system is not known to the programmer when the system is modeled.

An explicit model requires the programmer to explicitly declare which two tasks can be executed concurrently. This requires knowledge of the concurrency structure, or in this case the object structure, of the system. On the other hand, using an implicit concurrency model leaves concurrency decisions to the compiler and run-time system. When they are invoked, the object structure of the system is known. This makes our choice quite obvious:

Design choice 6. The concurrency model will be implicit.

Since the concurrency model will target real-time systems, it needs a notion of timing constraints. The timing model used in the Timber programming language [Lindgren et al., 2005], discussed in section 2.6, is deemed sufficient and it will therefore be used in our concurrency model. The baseline construct also permits delayed and periodic reactions in a simple and intuitive way, as discussed in section 2.6.

3.4 Formal definition

First, a set of operators needs to be defined. The first operator is the semicolon (;) that separates two elements (code statements, code blocks or the like) that will be executed sequentially. The statement to the right of the semicolon may only be executed when the statement to the left of it has finished. The second is the square brackets ([and]), which are used to denote an optional element.

Each system S is completely defined by the set of input events $\mathbf{I}$, output events $\mathbf{O}$, state variables $\mathbf{V}$, sequential execution blocks $\mathbf{E}$, reactions $\mathbf{R}$, reactions chains $\mathbf{C}$ and the functional mapping between inputs and reaction chains F.

$$S =< \mathbf{I}, \mathbf{O}, \mathbf{V}, \mathbf{E}, \mathbf{R}, \mathbf{C}, F > \tag{3.1}$$

Each reaction $R \in \mathbf{R}$ contains the initial sequential execution block $B \in \mathbf{E}$, possibly followed by a new reaction, in which case we will speak about a sequential call between the block and the subsequent reaction.

If multiple following reactions are defined, only one may be chosen as the continuation. Which reaction that is executed may depend on the result of calculations in B or the values of some state variables. In the following definition, the right arrow operator ($\rightarrow$) is used to denote that for each value a different R may be defined.

$$\begin{aligned} R &= B_i \\ &\mid B_i; R \\ &\mid B_i;\ \text{if } \textit{condition} \text{ then } R_j \text{ [else } R_k] \\ &\mid B_i;\ \text{while } \textit{condition}\ R \\ &\mid B_i;\ \text{case } \textit{variable} \text{ of } \{\text{value } \rightarrow R_j\}. \end{aligned} \tag{3.2}$$

A sequential execution block B is completely defined by its name, the code it contains, references to other blocks following it and references to all the state variables it contains,

$$B =< \text{name}, \text{args}, \text{code}, \text{Ref }_B, \text{Ref }_V >, \tag{3.3}$$

where *args* is a set of arguments to the code block and the code consists of assignments to either local variables or state variables, synchronous or asynchronous calls to other blocks, and producing outputs from the system as a whole. Ref_B is a subset of all executable blocks $\mathbf{E}$ and Ref_V is a subset of all state variables $\mathbf{V}$.

$$\begin{aligned}
\text{code} &= K;\text{code}\\
K &= V = \textit{expression}\\
&\mid\ L = \textit{expression}\\
&\mid\ v \leftarrow \text{sync } B \in \mathbf{E}\\
&\mid\ [\text{after } \textit{baseline-offset}]\ [\text{before } \textit{deadline}]\ \text{async } R \in \mathbf{R}\\
&\mid\ \text{output.}
\end{aligned} \tag{3.4}$$

Here, V is a state variable (preserved inbetween invocations and possibly shared by several execution blocks), L is a variable local to the code block and *expression* is any expression that may include arguments, local variables, state variables, and any pre-defined operators and function[2] calls.

The *sync* statement is a synchronous call to another code block, optionally returning a value. By allowing only a single code block to be called synchronously in contrast to calling an entire reaction, we ensure that the code block called synchronously cannot execute any sequential calls; only synchronous and asynchronous calls are allowed from it.

The *async* statement is an asynchronous call to another code block, which can be executed concurrently with the active block, and may include additional timing constraints.

A reaction chain C is iteratively defined as

$$C = \{R_{\text{start}}\} \cup \{R_i \mid \exists R_j \in C, R_i \in \text{invoked}(R_j)\} \tag{3.5}$$

where *invoked()* returns all reactions invoked, either asynchronously or synchronously, from the execution blocks comprising a given reaction, and R_{start} is the reaction triggered by an external event.

Let us now define in which case two blocks can be executed concurrently. If two sequential executions blocks B_i and B_j can be executed concurrently, they cannot belong to the same reaction and any two reactions that they belong to cannot have shared state variables,

[2]*function* is a purely functional calculation that produces the same output for the same input, does not have a state and does not produce any side effects.

$$\begin{aligned} B_i \| B_j \quad &\Leftrightarrow \quad i \neq j \\ &\wedge \quad \forall R \in \mathbf{R} : B_i \in R \Rightarrow B_j \notin R \\ &\wedge \quad SV^*(B_i) \cap SV^*(B_j) = \emptyset \end{aligned} \tag{3.6}$$

where

$$SV^*(B_k) = \cup \{SV(B_m) \,|\, \exists R : \; B_k \in R, B_m \in R\} \,. \tag{3.7}$$

$SV(B_m)$ returns all state variables used by a single execution block, and $B \in R$ means that the execution block is a part of the reaction R.

Next, the question of when two reactions can be executed concurrently must be addressed. Two reactions may only execute concurrently if

$$\begin{aligned} R_i \| R_j \quad &\Leftrightarrow \quad i \neq j \\ &\wedge \quad SV^*(R_i) \cap SV^*(R_j) = \emptyset \end{aligned} \tag{3.8}$$

where

$$SV^*(R) = \cup \{SV(B_i) \,|\, B_i \in R\} \tag{3.9}$$

Lifting the notion of concurrency from the execution code block level to the reaction level ensures that an execution block that is not allowed to execute concurrently with two blocks in another reaction, is not allowed to execute *in between* them. In this way we preserve a reaction (a sequence of sequential execution blocks connected by sequential calls) as a single unit of concurrency.

We will see later that, for a language with object-level concurrency like Timber, this transforms quite nicely into object level concurrency, by keeping any reaction (but not a reaction chain) confined to a single object.

An asynchronous call between two code blocks is always allowed, however, they may only execute concurrently if allowed by (3.6). A synchronous call is also always allowed, however, the target block may not call any other code block sequentially, as follows from the definitions of the synchronous call in (3.4).

CHAPTER 4
Graphical notation

4.1 Program flow and code blocks

The program flow is from left to right along vertical and horizontal arrows. External events (interrupts) are represented to the far left while the outputs are on the far right.

Each block of sequential code is represented by a square as in figure 4.1, it has inputs and outputs and may call other blocks.

The specific position of a code block in relation to other blocks does not imply any specific timing behavior; a code block may perform calls to all other code blocks, even blocks positioned to the left of it, as illustrated in figure 4.2.

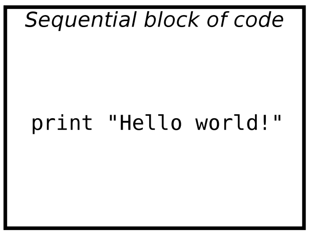


Figure 4.1: One sequential code block.

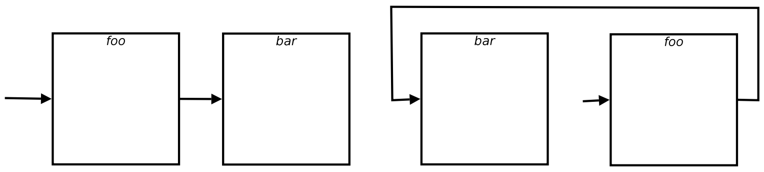
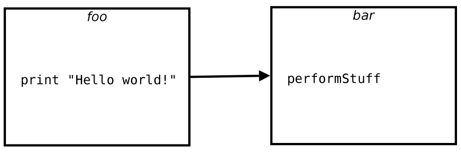
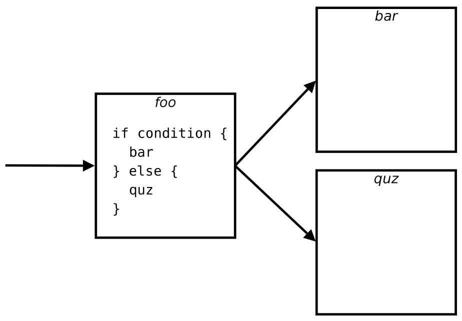


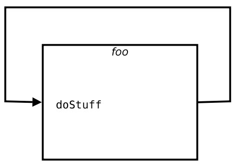
Figure 4.2: *foo* is always executed before *bar*, even if *foo* is placed to the right.



(a) One call is executed when block *foo* has run to end.



(b) *foo* performs a branch operation, selecting one of two possible routes.



(c) *foo* performs a loop or a recursive call.

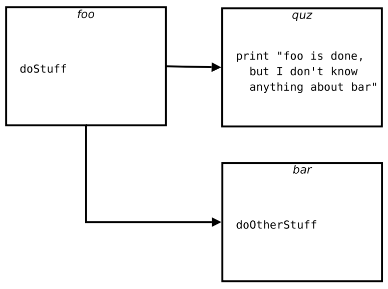
Figure 4.3: Sequential calls.

4.2 Interactions between code blocks

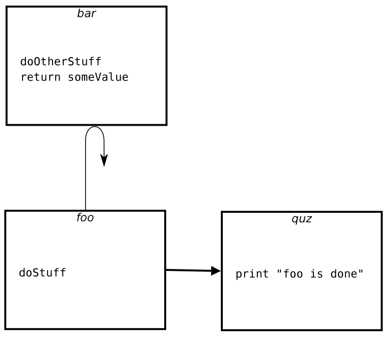
A sequential call is represented as a straight arrow exiting the code block on the right side as in figure 4.3(a). A code block performing a sequential call to itself, i.e. loops and recursive calls, is represented as an arrow exiting from the right side and looping around the block to the left side as in figure 4.3(c).

If more than one call is possible when a sequential code block has run to end, it is represented by more than one arrow exiting from the right side as in figure 4.3(b). Not more than one of these is chosen at each invocation.

An asynchronous call is represented by a straight arrow exiting the code block on the bottom or top side as in figure 4.4(a). A synchronous call is represented by a hooked arrow as in figure 4.4(b).



(a) Block *foo* starts *bar* concurrently with itself.



(b) Block *foo* performs a synchronous call to *bar*.

Figure 4.4: *foo* performing asynchronous and synchronous calls.

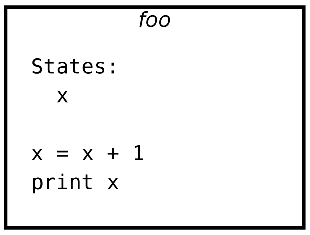
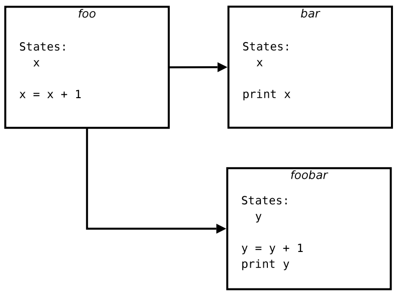


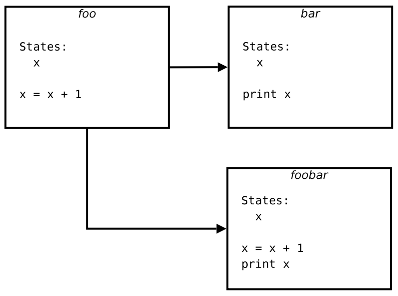
Figure 4.5: A sequential code block with a state variable.

4.3 State variables

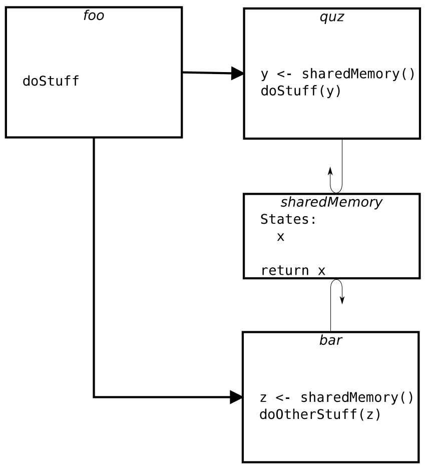
Figure 4.5 displays the notation for a sequential code block using a state variable. Figure 4.6 demonstrates the notation for sharing a state variable; in 4.6(b), the two reactions sharing x cannot execute concurrently. A solution allowing for concurrent execution is presented in figure 4.6(c).



(a) x is only used in one reaction of the two reactions forming a single reaction chain; *foobar* may execute concurrently with either *foo* or *bar*, or between them.



(b) x is used in two reactions forming a single reaction chain; *foobar* cannot execute until the first reaction, containing *foo* and *bar*, has completed.



(c) Read access to x is synchronized, thereby allowing concurrent execution.

Figure 4.6: Sharing a state variable.

CHAPTER 5
User interface

5.1 Language, libraries and design

Java (with the Swing GUI package) was chosen as the language for the user interface and backend from the start. To facilitate program development, *NetBeans*[1] was used.

The program was written using a *model-view-controller* architectural pattern. Program logic was separated from the user interface in a clean way as to make modification of the interface easier.

The program will be called the *Graphical Open Timber Helper* or GOTH for short. This symbolizes ease of use and a graphic, user friendly interface (*graphical*), open-source (*open*), based upon the Timber programming model (*Timber*) and shows the main point of the program: making life easier for the user (*helper*).

5.2 Main window

5.2.1 Input view, code block view and output view

The main window of the program, shown in figure 5.1, is divided into three areas. To the left, the program inputs are displayed and to the right are the program outputs. The larger view in the middle is where the code blocks and calls are displayed.

[1]An integrated development environment for Java by Sun Microsystems.

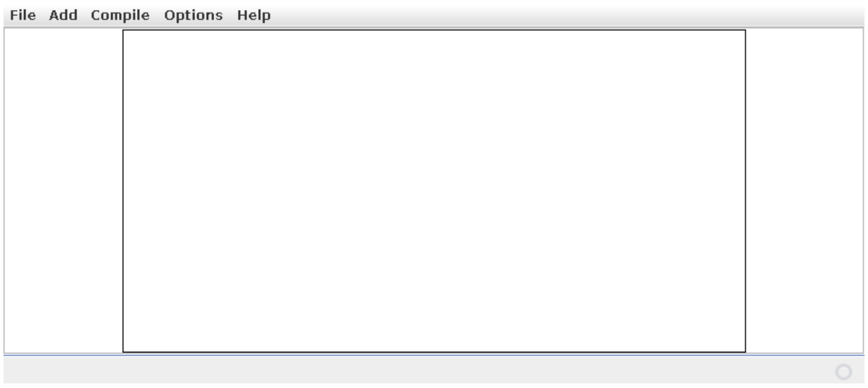


Figure 5.1: The program main window at startup.

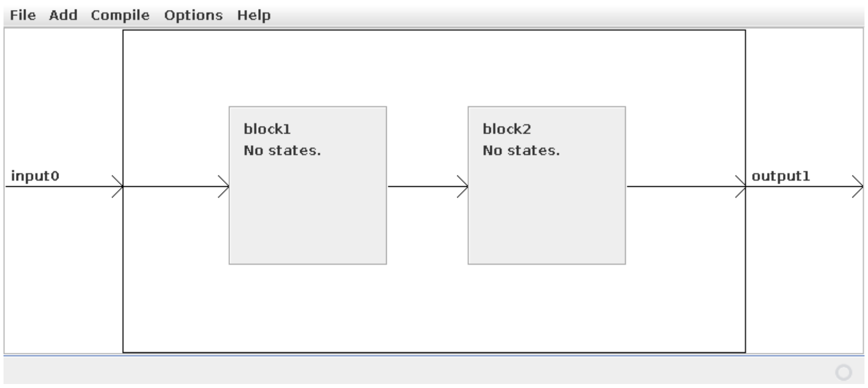


Figure 5.2: The program main window with an input, two code blocks and an output.

5.2.2 Program inputs and outputs

A program input is shown as a straight arrow extending from the left side of the window, across the input view to the code block view as in figure 5.2. Above the arrow, the name of the input is displayed. By holding the left mouse button, the input can be moved up and down to be placed at the perfect location as determined by the user. By right-clicking the input, the user can change the name as well as connect it to a code block and remove the program input entirely.

The program outputs are similar to the program inputs, except the arrow extends from the code block view, across the output view and ends at the right side of the program window.

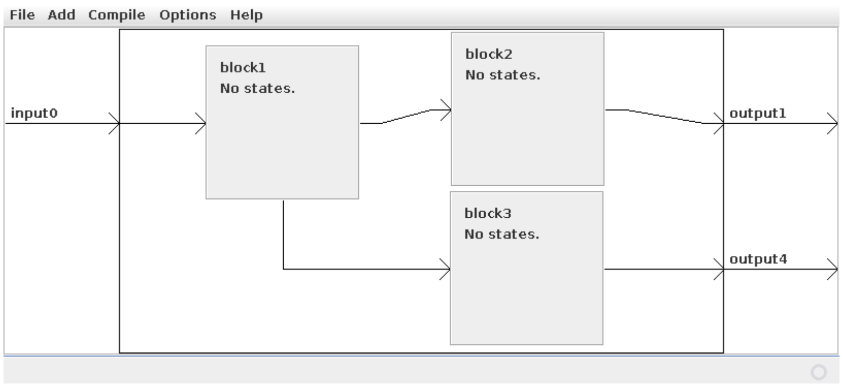


Figure 5.3: User has added an asynchronous call.

5.2.3 Code blocks

The code blocks are displayed as grey squares, a few centimeters on each side. They display the name of the code block, the input parameters and the state variables used by the block. Just as the program inputs and outputs, they can be dragged by using the left mouse button and their size can be changed by holding the left mouse button while grabbing one of the sides or corners.

By right-clicking a code block a pop-up menu is displayed. The user can then display and change all the properties of a code block, including input parameters and state variables as well as creating loops to itself or calls to other blocks. The block can also be connected to program inputs and outputs and also removed entirely.

5.2.4 Calls

Calls are displayed as black arrows, extending from the source (either a program input or a code block) to the destination (either a program output or another code block). As defined in the graphical notation, sequential calls extend from the right side and asynchronous and synchronous calls extend from the top or bottom side.

In figure 5.3, the user has created a program with one input and two outputs. One of these outputs, *output1*, is the output from the first sequential chain of code blocks, but *output4* is the result of an asynchronous call to *block3*.

Figure 5.4 displays a program with one sequential call (top), one synchronous call (middle) and one asynchronous call (bottom code block).

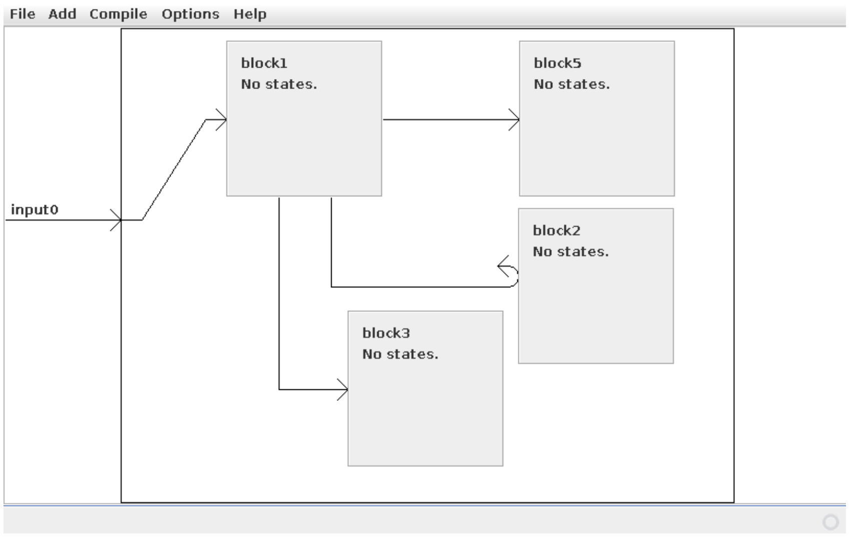


Figure 5.4: All three different call types.

5.3 Data structures

Six different structures were used to store information, each implemented in a separate Java class.

The parent class is *Program* that keeps track of, among other things, all code blocks in the program. The *CodeBlock* keeps track of its inputs and outputs, as well as its state variables, parameters and other internal states.

Each variable and parameter is an instance of the *Variable* class.

The code block inputs and outputs as well as the program inputs and outputs are instances of the *InputOutput* class. They create calls between them that are instances of the *Call* class.

Call stores the call type and the base- and deadline associated with the call. Baselines and deadlines are instances of the *Time* class.

5.3.1 Program

The program keeps track of its name, all code blocks and the program inputs and outputs. Each program also creates its own backend. The internal structure is:

- *codeBlocks*, all code blocks in the program
- *inputs*, all program inputs
- *outputs*, all program outputs

- *name*, the name of the program
- *backend*, the backend that creates the code skeleton

5.3.2 codeBlocks

Each codeblock keeps track of its inputs, outputs, state variables and its input parameters as well as its name and to which program it belongs. An internal ID that is assigned when the codeblock is created is used to check if this is the same as another code block. The internal structure is:

- *name*, the name of the code block
- *inputs*, the inputs to the code block
- *outputs*, the outputs to the code block
- *variables*, the internal state variables
- *parameters*, the input parameters
- *parentProgram*, the program to which the code block belongs
- *ID*, an identifier used for comparing this to other code blocks

5.3.3 Inputs and outputs

The inputs and outputs have their own type, *InputOutput*. They keep track of up to two calls and whether it belongs to a codeblock or to the program itself as well as if it is an input or an output. The internal structure is:

- *call*, call to another code block or program input or output
- *extCall*, call to or from the external world
- *parentBlock*, parent code block (if applicable)
- *parentProgram*, parent program (if applicable)
- *blockOrProgram*, whether this belongs to a code block or program
- *inputOrOutput*, whether this is in input or output
- *name*, the name, used for external inputs and outputs

5.3.4 Calls

The *Call* type represent an actual call between two *InputOutput*s. Each call keeps track of its origin and destination, as well as its type, parameters, baseline and deadline. The internal structure is:

- *callFrom*, origin for this call
- *callTo*, destination for this call
- *callType*, whether this call is asynchronous, synchronous or sequential
- *parameters*, the parameters that are sent with this call
- *after*, baseline for this call
- *before*, deadline for this call

5.3.5 Variable

Each variable keeps track of its name and which codeblock it belongs to. The internal structure is:

- *varName*, the name of this variable
- *parentBlock*, the code block this variable belongs to

5.3.6 Time

The time structure is used to store base- and deadlines. It stores the time amount, the time unit and whether this base- or deadline is inherited or not. If it is inherited, the time is ignored. The internal structure is:

- *time*, the time amount
- *unit*, the time unit
- *inherit*, if this base- or deadline is inherited

5.4 Hand-over to backend

When the user selects *Compile*, the backend is initialized. The *Program* structure is handed over directly to the backend and with it all related constructs such as code blocks, calls and state variables. The backend parses the structure to generate a code skeleton.

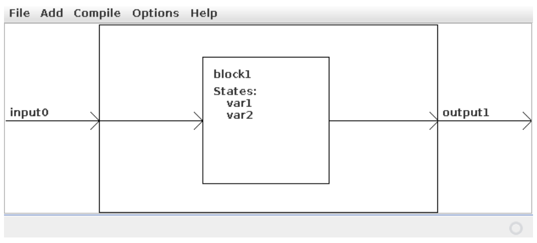


Figure 5.5: A very simple program.

5.5 Examples

5.5.1 A simple program

One of the simplest implementation of our concurrency model is a real-time system with only one input, one output and one place to insert code. An example might be a system lighting a LED in response to a button press.

Starting with the main window as displayed in figure 5.1, right click in the middle of the window and select *Add code block*. A code block called *block1* now appears under the mouse pointer.

Right click in the left, input window and select *Add*. A program input called *input0* now appears. To create the program output, right click in the right, output window, and select *Add*. A program output called *output0* appears under the mouse pointer.

The input, output and code block can be dragged by using the mouse to place them as aesthetically as possible.

To connect the program input to the code block, right click on the input. Select *Connect to...* → *block1*. The output is connected in much a similar way by right clicking on it and selecting *Connect from...* → *block1*.

Now create two state variables in *block1*. This is done by right clicking on the code block and selecting *Add variable...*. The default names will do for now but you are free to name them anything you want. Just remember that two variables with the same name are treated as the *same* variable.

The finished result is shown in figure 5.5 and the code skeleton generated for this design using the Timber backend is found in appendix A.1.

5.5.2 A sonar program

Another, a bit more complex example is a *sonar*. A sonar sends out a signal and waits for it to come back. The time in between is used to compute the distance from the sonar to the object. If no reflection comes back within a certain time frame, no objects are assumed to be in front of the sonar.

This program will have two external inputs; one for the signal to activate

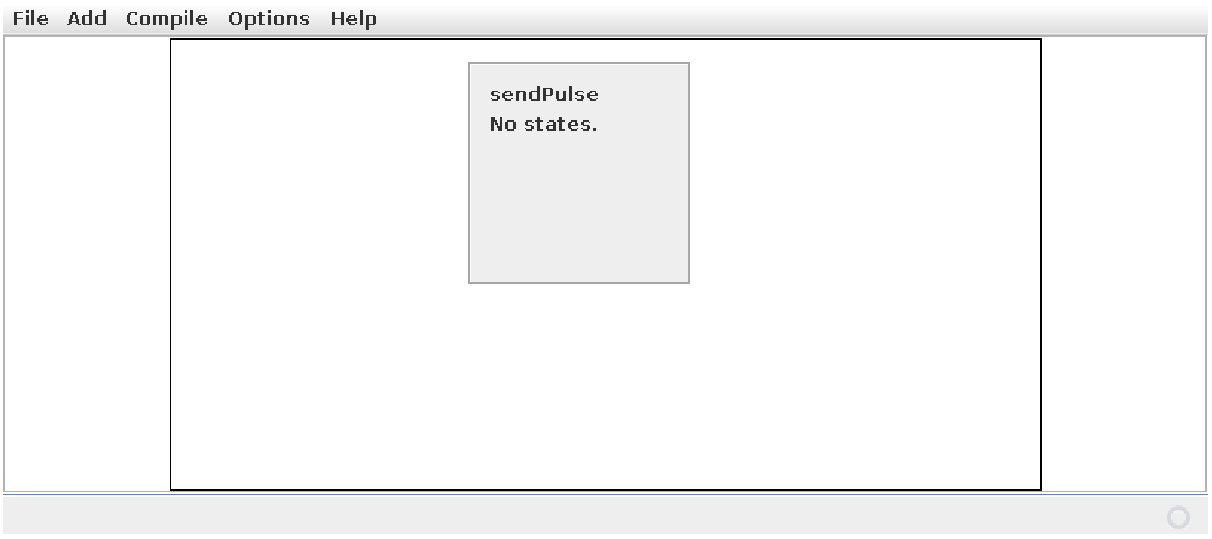


Figure 5.6: The first code block has been added to the sonar program.

the sonar and one for receiving the reflected signal. Similarly, three outputs are needed; one for triggering a sonar pulse, one for returning the distance measurement and for reporting a timeout. Seven code blocks need to be defined:

- one for generating a sonar pulse when the activation input is triggered,
- one for computing the distance when the reflected pulse is received,
- one responsible for the timeout when a signal is not received,
- and four for storing and setting the pulse time and system state.

Starting with the main window as displayed in figure 5.1, right click in the middle of the window and select *Add code block*. A block called *block1* now appears under the mouse pointer.

Since *block1* is not a very descriptive name, rename the block by right clicking on it and selecting *Properties...*. Change the name to *sendPulse* and press the OK button. Move the code block to a good spot in the window. Your program should now look somewhat like figure 5.6.

Now repeat this for another six code blocks that are to be called *calculate-Distance*, *handleTimeout*, *saveState*, *getState*, *saveTime* and *getTime*. You may have to resize the program window to fit all the blocks. Refer to figure 5.7 for the end result.

Add the two program inputs by right clicking in the left, input part of the window and selecting *Add*. A new program input appears under the mouse

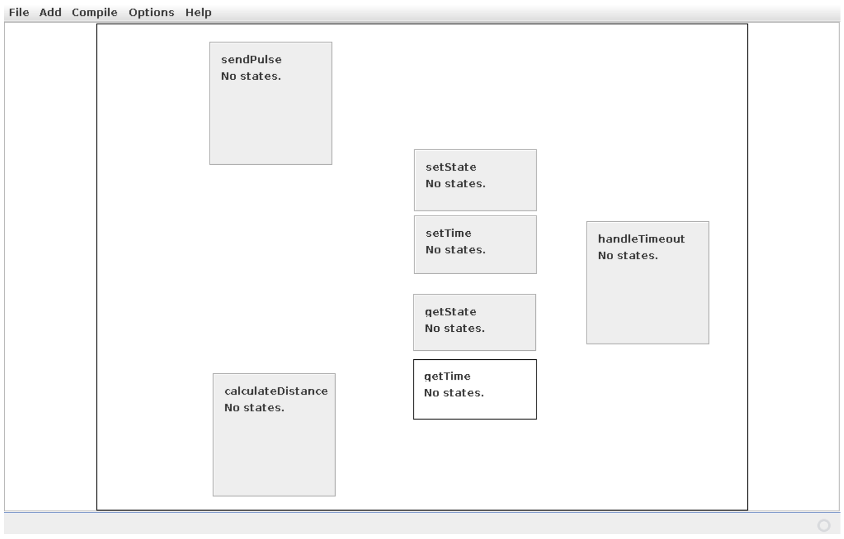


Figure 5.7: All seven blocks have been added to the sonar program.

pointer. Rename this to *triggerInput* by right clicking on the arrow, selecting *Properties...*, entering the new name and clicking OK.

Move the input to the left of *sendPulse* and repeat the step above to add another new input called *pulseInput* to the left of *calculateDistance*.

Similarly, add the three program outputs by right clicking in the right, output part of the window and selecting *Add*. Rename the output to *pulseOutput* and place it to the right of *sendPulse*. Add another output called *distanceOutput* to the right of *calculateDistance* and one called *timeoutOutput* to the right of *handleTimeout*. Your program should now look somewhat like figure 5.8.

Now, the inputs and outputs need to be assigned to code blocks. Assign *triggerInput* to *sendPulse* by right clicking on the input arrow and selecting *Connect to...* → *sendPulse*. Similarly, assign *pulseInput* to *calculateDistance*.

Assign *sendPulse* to *pulseOutput* by right clicking on the output and selecting *Connect from...* → *sendPulse*. Assign *handleTimeout* to *timeoutOutput* and *calculateDistance* to *distanceOutput* in a similar way.

The code blocks now need some internal state. *getState* and *setState* will need a shared state variable called *systemState* to store the system state, whether it is *idle* or *waiting for reflection*. *getTime* and *setTime* will need a state variable called *timeSent* to store the time when the pulse was sent.

Create two state variables by right clicking on the block called *setState* and selecting *Add variable...*. Name the variable *systemState*. Create another variable called *systemState* in *getState* and a state variable called *timeSent*

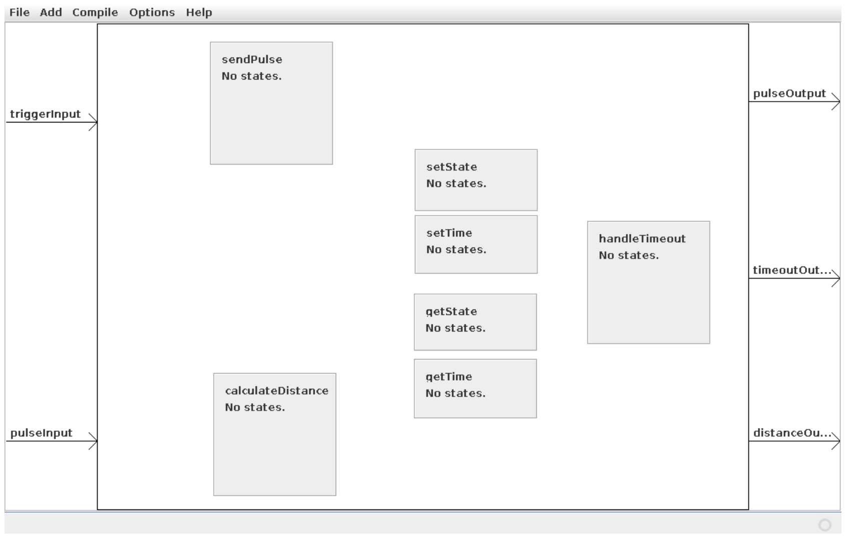


Figure 5.8: The inputs and outputs have been added to the sonar program.

in both *getTime* and *setTime* in a similar way.

The next step is to define the interactions between the code blocks. Add a synchronous call from *sendPulse* to *setState* by right clicking on *sendPulse* and selecting *Add call to...* → *setState* → *synchronous*. An arrow now appears between the code blocks. Add another synchronous call to *setTime*.

Add a synchronous call from *calculateDistance* to *getState* and one to *getTime*, as well as a synchronous call from *handleTimeout* to *setState*.

handleTimeout should be invoked a certain time after a pulse has been sent but no reflection received. Add an asynchronous call from *sendPulse* to *handleTimeout* by right clicking on *sendPulse* and selecting *Add call to...* → *handleTimeout* → *asynchronous*. Your program should now look like figure 5.9.

handleTimeout should not be invoked until a certain time has elapsed. Right click on the call between the blocks and check the *After* checkbox. Enter one second as the time and press OK. The invocation of *handleTimeout* will now be delayed one second.

Now select *Compile* in the *Compile* menu. Change the path and run time system as you see fit or leave it as the defaults. Press *Begin*. When the process has finished, check the directory you specified. There should be six files there called *FooProg.t* and *FooClassn.t* where *n* ranges from zero to four. A log file, *FooProg.log* is also created. The source code can be found in appendix A.2.

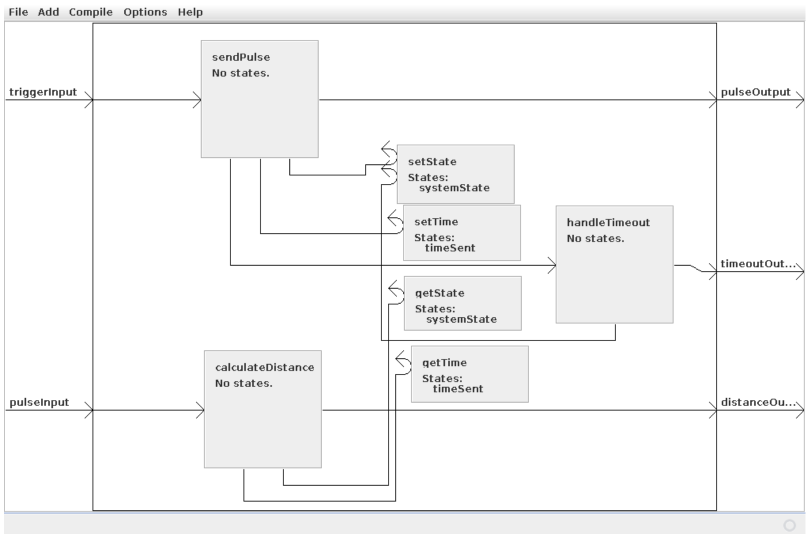


Figure 5.9: All calls have been defined in the sonar program.

CHAPTER 6
Timber backend

This chapter describes the transformation of a graphical model into a code skeleton in the Timber language.

6.1 Object model

The concurrency model needs to be expressed in Timber code in a way that captures the desired functionality. However, the question remains whether the Timber backend may disallow constructs that are allowed in the concurrency model already defined. Disallowing certain constructs that are difficult to express in Timber code may be considered as defeating two of the aims of the graphical design tool: user-friendliness and making the backend easily replaceable.

Design choice 7. A backend should not disallow constructs allowed by the concurrency model.

6.1.1 Sequential calls

Since in Timber any two objects can execute concurrently and there is no pre-defined ordering of action calls made to the same object, the only way to conform to our definition of concurrent reactions is to make each sequential call into a *do*-construct. This way, the Timber run-time system never releases the lock on the object when executing a sequential call. Additionally, in some cases a sequence of sequential calls can be merged into one method.

Design choice 8. A sequential call will always be expressed as a *do*-construct or by merging the two code blocks into the same method.

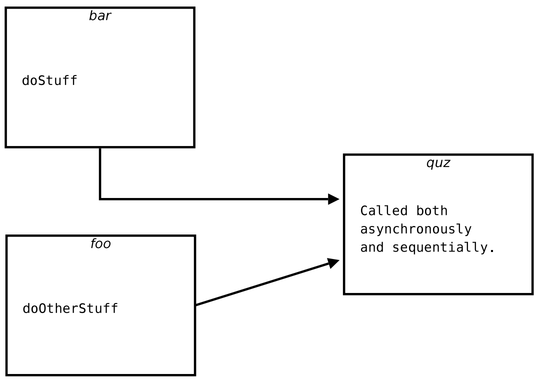


Figure 6.1: A code block called both asynchronously and sequentially

6.1.2 Asynchronous calls

Asynchronous calls between two code blocks a and b can be expressed in two ways: either keeping them in the same object or putting them in separate objects. In order to maximize parallelism, the two code blocks should be placed in separate objects. If, however, they share a state variable, they are not allowed to execute concurrently according to (3.6) and they are therefore forced into the same object.

Design choice 9. Asynchronous calls between two code blocks will force them into different objects if they are allowed to execute concurrently. Otherwise they will be forced into the same object.

This leads to an interesting problem: what if a code block is called *both* sequentially and asynchronously as in figure 6.1? The problem has three solutions:

1. Make both calls to quz asynchronous. Both calls will be preemptible and the backend needs to decide whether foo, *bar* and quz should be in the same object or not.
2. Make foo call quz sequentially and *bar* call quz asynchronously. foo and quz are then forced into the same object and *bar* may be in a different object.
3. The construct is not allowed.

Our concurrency model allows this scenario, as quz is allowed to execute concurrently with *bar* but not with foo according to (3.6). Thus, solution 3 is ruled out according to design choice 7. Solution 1 would break a single reaction into a reaction chain and thus does not conform to our concurrency model.

Design choice 10. If a code block is called both sequentially by *a* and asynchronously by *b*, it will be forced into the same object as *a* and a different object as *b*. An asynchronous wrapper will be used for calls from *b*.

This wrapper is expressed in Timber code as

<within some class definition>

```
fooBlock = do
  -- Code for fooBlock goes here
fooBlock_wrapper  = action
  fooBlock
```

6.1.3 Synchronous calls

Without shared state variables

A synchronous call between two code blocks *a* and *b* can be expressed in two ways in Timber code:

1. *a* and *b* are put in the same object. To avoid deadlock, *b* is forced to become a *do*-construct.
2. *a* and *b* are put in different objects. *b* is then forced to become a *request*-construct.

According to our concurrency model, the two solutions are equivalent. However, the second solution incurs more overhead than the first.

Design choice 11. A synchronous code block which is only called by other code blocks that are all in the same object is made into a *do*-construct and put in the same object.

If two code blocks *a* and *b* call *c* synchronously, three solutions are possible:

1. *a*, *b* and *c* are put in different objects. *b* is made a *request*.
2. *a* and *c* are put in the same object and *b* in a different. *c* is made a *do*-construct and a *request* wrapper is used for calls from *b*.
3. *b* and *c* are put in the same object and *a* in a different. *c* is made a *do*-construct and a *request* wrapper is used for calls from *a*.

While all solutions conform to our concurrency model, the first one allows for a greated potential parallelism. Hence,

Design choice 12. If two or more code blocks that may belong to different objects call a code block synchronously, the code block is put in a separate object using a *request*-construct.

With shared state variables

If two code blocks *a* and *b* in different objects each call *c* synchronously, as in figure 6.2, but *a* and *c* have shared state variables, design choice 12 does not apply since state variables cannot be shared across objects.

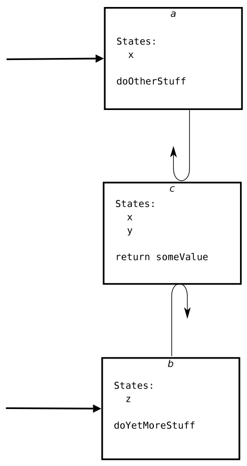


Figure 6.2: Synchronous call with a shared state variable

Three solutions are possible:

1. Put *a*, *b* and *c* all into the object. *c* is then expressed as a *do*-construct.
2. Put *a* and *c* into the same object and *b* in a different. *c* is expressed as a *do*-construct and a synchronous wrapper is used for calls from *b*.
3. Disallow the construct.

The first solution seems overly restrictive and would force the entire reactions including *a* and *b* into the same object. This may not be what the programmer intended. The third solution is ruled out because all constructs should be expressible. The second solution was therefore chosen.

Design choice 13. If code blocks in different objects call a code block synchronously and share state variables, all code blocks sharing state variables are forced into the same object. The other code blocks are put in separate objects and the synchronous calls are made via a *request* wrapper.

6.2 Code generation

When the user selects *Compile* in the user interface, an instance of the `TimberBackend` class is created. It is then handed the entire `Program` structure and the backend then parses it to generate a code skeleton.

6.2.1 Generation of objects

First, the system is partitioned into objects. All code blocks using shared state variables or connected via sequential calls are put into the same object. When partitioning is done, all remaining code blocks are put into separate objects to maximize parallelism, by design choice 3. If a reaction chain is split into multiple objects, this also permits multiple instances of the same reaction chain (but not of the same reaction) to be executed concurrently.. Each object is an instance of the `TimberClass` class.

6.2.2 Generation of methods

The objects are then partitioned into methods. Currently, each code block becomes a new method, but future work could include merging code blocks into larger methods to improve system performance. Each method is an instance of the `TimberMethod` class.

6.2.3 Generation of the root class

The backend then generates the root class. Code for each object is generated by each `TimberClass`, based upon the code generated by each `TimberMethod`.

The user selects a run-time system which is an instance of the `TimberRTS` class that generates headers for the specific RTS. Each class is then written to a separate file.

CHAPTER 7
Evaluation

7.1 Choice of language and libraries

Developing the program in the Java programming language enabled the program to be more platform independent than what otherwise would have been possible. This proved to be an advantage since the development and testing was done on both Linux and Mac OS X platforms.

Using the Swing toolkit for the user interface proved to be by far the most complex part of the program. Swing does what it is intended to do very well, however, if you want to do something else, you are mostly on your own.

Some parts of the program needed to be implemented from scratch in Swing. This included the drawing area, the graphical representation of calls and code blocks as well as inputs and outputs. This turned out to be more complex than first anticipated.

7.1.1 Interoperability problems

Some problems were found when running the program on another platform than on which it was designed. This was mostly because of what Swing calls different *look and feels*. The look and feel specifies such things as how windows should look and behave, how buttons should be placed and what font to use.

It turned out that different look and feels included different widgets in dialogs. For example, the *File open*-dialog in Mac-OS X looked very different from the one in Linux, causing some problems.

This was solved by including a menu in which the user could choose which look and feel he or she preferred.

7.2 User interface

Designing an easy to use but at the same time flexible and powerful user interface proved a challenge. Several designs were tried and evaluated before the final design was chosen.

A user interface where the user could place the components where he or she deemed fit was chosen over a more strict interface where the program itself decided how the components were to be placed. This gave the user more freedom in designing the program.

No usability study for the interface has been done; studying user responses and improving the interface is left for future work.

7.2.1 Flat versus hierarchical model

In this thesis, a flat model for combining blocks of code was chosen, primarily due to time constraints when developing the concurrency model and the backend. A hierarchical model would be preferable, but is left for future work.

7.3 Timber backend

The Timber backend was from the start written to be able to be easily replaced by both another backend for the Timber language and by a backend for some other language suitable for embedded, real-time systems. This proved to be a winning strategy; the backend needed to be rewritten and adapted to new scenarios as the concurrency model was being developed. If the backend had been tightly linked to the user interface, this would have been much harder. Backends for other languages are left for future work.

7.3.1 Object creation

The object creation was implemented by a quite inefficient algorithm, this could probably be replaced by a much simpler and faster algorithm quite easily. One example that could be fairly quickly implemented is

1. Generate a hash table of code blocks

2. Each code block and variable has an object-identifier (and is aware of which object it belongs to which is initialized to null).

3. Iterate over all code blocks in the program. For each code block:
   (a) Set all variables to a new object x.
   (b) If any variable already belongs to some object y:
      i. Set all variables belonging to object y to object x.
4. Iteration is complete when the last code block has been processed.
5. Each code block belongs to the same object as all its state variables.
6. Iterate over all code blocks in the program. For each code block:
   (a) Set this code block to a new object x.
   (b) Set all code blocks that *directly* follows this sequentially to object x.
      i. If any of them already belongs to some object y, set all code blocks belonging to object y to object x.
7. Iteration is complete when the last code block has been processed.

Implementing this, or some other algorithm, is left for future work.

APPENDIX A

Code examples using the Timber backend

A.1 Simple program with one code block

This program example has one input (*input0*), one output (*output0*) and one code block (*block1*) with two state variables (*var1* and *var2*).

A.1.1 FooProg.t

`FooProg.t` contains the root class that invokes all other classes. In this case, only one class (*fooClass0*) which describes the code block *block1*, is instantiated.

```
module FooProg where

  import POSIX
  import FooClass0

  root :: Env -> Class Action
  root env = class
    fooObj0 = new fooClass0 env

    start = after (millisec 50) before (millisec 50) action
      -- Start code goes here
    result start
```

A.1.2 FooClass0.t

```
module FooClass0 where

  import POSIX

  struct FooClass0 where
    block1 :: Action

  fooClass0 env = class
    var2 := value
    var1 := value
    block1 = do
      -- Code for block block1 goes here.
      env.output1
    block1_wrapper  = action
      block1
    result
      FooClass0{
        block1 = block1_wrapper
      }
```

A.2 A sonar program

Example code for a sonar program with two inputs, three outputs and seven code blocks. The system sends a sonar pulse whenever it is triggered, waits for a reflection and computes the distance. If no pulse is received for a certain time, in this case one second, a timeout is triggered.

A.2.1 FooProg.t

`FooProg.t` is the root class that creates the other objects.

```
module FooProg where

  import POSIX
  import FooClass0
  import FooClass1
  import FooClass2
  import FooClass3
  import FooClass4

  root :: Env -> Class Action
  root env = class
    fooObj0 = new fooClass0 env
    fooObj1 = new fooClass1 env
    fooObj2 = new fooClass2 env fooObj4 fooObj1 fooObj0
    fooObj3 = new fooClass3 env fooObj1 fooObj0
    fooObj4 = new fooClass4 env

    start = after (millisec 50) before (millisec 50) action
      -- Start code goes here

    result start
```

A.2.2 FooClass0.t

`FooClass0.t` contains the code for the shared *systemState* variable. It has two functions, *setState* and *getState*.

```
module FooClass0 where

  import POSIX

  struct FooClass0 type0 type1 where
    setState :: Request type0
    getState :: Request type1

  fooClass0 env = class
    systemState := value
    setState = do
      -- Code for block setState goes here.
    setState_wrapper = request
      foo <- setState
      result foo
    getState = do
      -- Code for block getState goes here.
    getState_wrapper = request
      foo <- getState
      result foo
    result
      FooClass0{
        setState = setState_wrapper,
        getState = getState_wrapper
      }
```

A.2.3 FooClass1.t

`FooClass0.t` contains the code for the shared *timeSent* variable. It has two functions, *setTime* and *getTime*.

```
module FooClass1 where

  import POSIX

  struct FooClass1 type0 type1 where
    getTime :: Request type0
    setTime :: Request type1

  fooClass1 env = class
    timeSent := value
    getTime = do
      -- Code for block getTime goes here.
    getTime_wrapper = request
      foo <- getTime
      result foo
    setTime = do
      -- Code for block setTime goes here.
    setTime_wrapper = request
      foo <- setTime
      result foo
    result
      FooClass1{
        getTime = getTime_wrapper,
        setTime = setTime_wrapper
      }
```

A.2.4 FooClass2.t

`FooClass1.t` implements the *sendPulse* code block.

```
module FooClass2 where

  import POSIX
  import FooClass4
  import FooClass1
  import FooClass0

  struct FooClass2 where
    sendPulse :: Action

  fooClass2 env fooClass4 fooClass1 fooClass0 = class
    sendPulse = do
      -- Code for block sendPulse goes here.
      after (sec 1) fooClass4.handleTimeout
        foo <- fooClass1.setTime
        foo <- fooClass0.setState
      env.output3
    sendPulse_wrapper  = action
      sendPulse
    result
      FooClass2{
        sendPulse = sendPulse_wrapper
      }
```

A.2.5 FooClass3.t

FooClass2.t contains the code block for calculating the distance when a signal is received.

```
module FooClass3 where

  import POSIX
  import FooClass1
  import FooClass0

  struct FooClass3 where
    calculateDistance :: Action

  fooClass3 env fooClass1 fooClass0 = class
    calculateDistance = do
      -- Code for block calculateDistance goes here.
        foo <- fooClass1.getTime
        foo <- fooClass0.getState
      env.output5
    calculateDistance_wrapper  = action
      calculateDistance
    result
      FooClass3{
        calculateDistance = calculateDistance_wrapper
      }
```

A.2.6 FooClass4.t

`FooClass4.t` handles the timeout. It is invoked by *sendPulse* when a pulse is sent; if a pulse is received, it is aborted by *calculateDistance*.

```
module FooClass4 where

  import POSIX

  struct FooClass4 where
    handleTimeout :: Action

  fooClass4 env = class
    handleTimeout = do
      -- Code for block handleTimeout goes here.
      env.output4
    handleTimeout_wrapper  = action
      handleTimeout
    result
      FooClass4{
        handleTimeout = handleTimeout_wrapper
      }
```

Bibliography

Ada manual (2006). *Ada Reference Manual.* Ada working group. 3rd edition.

AdaCore (2009). Ada overview. `http://www.adacore.com/home/ada_answers/ada_overview/`.

Awad, M. & Ziegler, J. (1997). A practical approach to the design of concurrency in object-oriented systems. *Software – practice and experience*, 27(9), 1013–1034.

Berry, G. (1993). Preemption in concurrent systems. In *Foundations of software technology and theoretical computer science*, volume 761 (pp. 72–93). Springer.

Burns, A. & Wellings, A. (2001). *Real-Time Systems and Programming Languages.* Pearson Education, 3rd edition.

Henziger, T. A., Horowitz, B., & Kirsch, C. M. (2001). Giotto: A time-triggered language for embedded programming. In *Proceedings of the international workshop on embedded software* (pp. 166–184).: Springer.

Lee, E. A. (2006). *The problem with threads.* Technical report, University of California, Berkeley, USA.

Lee, E. A. & Neuendorffer, S. (2005). Concurrent models of computation for embedded software. *IEE proceedings: computers and digital techniques*, 152(2).

Lee, E. A. & Zhao, Y. (2007). Reinventing computing for real time. In *Reliable systems on unreliable networked platforms*, volume 4322 (pp. 1–25). Springer.

Levis, P. (2006). *TinyOS programming.* Revision 1.3.

Lindgren, P., Nordlander, J., Aittamaa, S., & Eriksson, J. (2008). Tinytimber, reactive objects in c for real-time embedded systems. In *Proceedings, Design, Automation and Test in Europe* (pp. 1382–1385).: European Design and Automation Association.

Lindgren, P., Nordlander, J., Svensson, L., & Eriksson, J. (2005). *Time for Timber*. Technical report, Luleå University of Technology, Luleå.

Nordlander, J., Jones, M. P., Carlsson, M., Kieburtz, R. B., & Black, A. (2002). Reactive objects. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*.

Sanchez, C., Sipma, H. B., Gill, C. D., & Manna, Z. (2006). Distributed priority inheritance for real-time and embedded systems. In *Principles of Distributed Systems* (pp. 110–125). Springer.

Scade manual (2007). *Scade language reference manual*. Esterel Technologies.

Schneider, F. B. & Andrews, G. R. (1986). Concepts for concurrent programming. In *Current trends in concurrency* (pp. 669–716). Springer.

Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., & Deprettere, E. (2004). System design using Kahn process networks: The Compaan/Laura approach. In *Proceedings of the conference on design, automation and test in Europe*, volume 1 (pp. 10340).: IEEE computer society.