Using Coroutines for Multi-core Preemptive Scheduling

Ole Lehrmann Madsen
ole.l.madsen@cs.au.dk
Aarhus University
Aarhus, Denmark

Abstract

The advent of multi-core processors has increased the demand for programming concurrent systems. In this paper, we explore the use of SIMULA style coroutines and other primitives as a basis for defining a broad class of high-level concurrency abstractions including the definition of associated schedulers. The main contribution in this paper is an implementation of preemptive coroutines for a multi-core processor in an experimental version of Beta. The overall goal is to use a high-level language to program applications on a bare bone platform without an operating system.

ACM Reference Format:
Ole Lehrmann Madsen. 2021. Using Coroutines for Multi-core Preemptive Scheduling. In *11th Workshop on Programming Languages and Operating Systems (PLOS '21), October 25, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3477113.3487271

1 Introduction

Concurrent[1] programming is inherently difficult and multi-core processors has further increased the requirements on languages and tools for concurrent programming.

Mainstream programming languages such as C [15], C++ [23], Java [10], and C# [12] are of limited help with respect to concurrent programming since they mainly offer low-level mechanisms such as threads, locks, and semaphores. This is despite the fact that many proposals for safe high-level concurrent programming languages have been made, including Concurrent Pascal [4], Actor-based languages [2, 13], like ABCL/1 [27], and Erlang [3], CSP [14], Ada [1], Concurrent Smalltalk [26], and many more. For most of these languages, the concurrency mechanisms are built into the language and there is limited support for defining new concurrency abstractions.

For a certain class of concurrent systems, you may want to have full control over the scheduling of concurrent processes – this includes processes running in true parallel (e.g. on different cores), being preemptively scheduled, using cooperative scheduling or a mixture of these. This is especially the case for embedded systems where you do not want to rely on an operating system. Here you may also want to avoid the overhead of an operating system with respect to time and space just as you may want to bypass the scheduling mechanism of the operating system.

The work presented here is part of a project where the *overall goal* is to design a language based on a few simple low-level primitives and powerful abstraction mechanisms that makes it possible to define *safe high-level concurrency abstractions*. This includes mechanisms for defining schedulers for a given concurrency model and the ability to implement these on a multi-core architecture. The ultimate goal is to be able to develop bare bone applications using safe high-level concurrency abstractions with full control over scheduling.

In this paper, we show how to implement preemptive schedulers for coroutines on a multi-core platform. The starting point is coroutines as found in SIMULA [7, 8] and further refined in Beta [20]. A coroutine is a cooperative thread in the sense that there is no preemption, but preemption of coroutines was later added in the Lund SIMULA system [24].

In [17], we described a further development of the coroutine and synchronization mechanisms of Beta in a variant of Beta called `xBeta`. The main goal of `xBeta` has been to be able to define *safe* concurrency abstractions in the sense of Brinch-Hansen [5].

In this paper, we use a further development of `xBeta` called `qBeta`.[2] At the basic level, `qBeta` (as Beta and `xBeta`) is unsafe and has only few simple primitives for handling concurrency, communication, and synchronization. As shown in [17], it is possible to define safe concurrency abstractions in the form of application frameworks.[3]

Using a safe concurrency framework, corresponds to programming at the level of Concurrent Pascal, Erlang, etc. If

[1] We use concurrent and parallel as synonymous terms.

PLOS '21, October 25, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8707-1/21/10…$15.00
https://doi.org/10.1145/3477113.3487271

[2] The differences between `xBeta` and `qBeta` are not important here.

[3] By application framework we refer to a module that defines a set of abstractions for a given concurrency model.

you use qBeta directly, this will correspond to the implementation level of say Erlang, Actors, etc. That is, in order to experiment with concurrency abstractions, scheduling policies, etc., in most cases for these languages you will have to stick to the level of C or C++. qBeta is in this sense a general purpose object-oriented language that supports implementation of safe high-level concurrency frameworks within the same language.

The implementation of xBeta presented in [17] is for a single-core processor. Here we show how to implement preemptive coroutines on a multi-core platform.

In [17], we showed how to define safe monitor and Ada-like rendezvous frameworks. In this paper, we show how to define a framework for implementing a notion of *Simple Concurrent Processes* (SCP). A subtle feature of this concurrency framework shows how to use coroutines to implement *asynchronous method invocation*. The main purpose of this framework is to be used as an example of how to define a higher-level concurrency abstraction based on coroutines. The SCP framework is thus not intended to be used for production programming.

Safety with respect to race conditions is a major goal of our work, and we have proposed the notion of *subpattern restrictions* [17] to ensure safety of a given concurrency framework. Space does not permit us to address subpattern restrictions in this paper, instead the reader is referred to [17] .

In summary, we show how to implement preemptive scheduling of coroutines on a multi-core platform; we present an example of a concurrency abstraction built on low-level language mechanisms; and we show how coroutines may implement asynchronous method invocation. A prototype compiler and VM have been implemented for qBeta and all examples presented here may be compiled and executed.

2 Basic language mechanisms

qBeta is a further development of Beta [16, 20] and xBeta [17] with focus on coroutines and concurrency. The syntax is inspired by Python [22] where nesting (block-structure) is defined by indentation. To save space, we sometimes use curly brackets (`{...}`) to describe block-structure and semicolons (`';'`) to separate statements. We assume that the reader is familiar with patterns, submethods and inner in Beta. An extended version of this paper with more details on qBeta and a short description of how to define a safe concurrency abstraction is available from Aarhus University [18].

In this section, we give examples of Beta submethods and a monitor[4] system is used to show how to define a concurrency abstraction. The monitor abstraction in the form of a class is shown in Figure 1.

Instances of the Monitor class have a Semaphore M. They have a method attribute entry that must be a supermethod of all public methods of a Monitor object. In Figure 1, it is

```
class Monitor:
  M: obj Semaphore
  void entry(): { M.wait; inner; M.signal }
  ...
buffer: obj Monitor
   L: obj List;
   void put(E: integer): entry{ L.put(E) }
   integer get(): entry{ return L.get() }
```

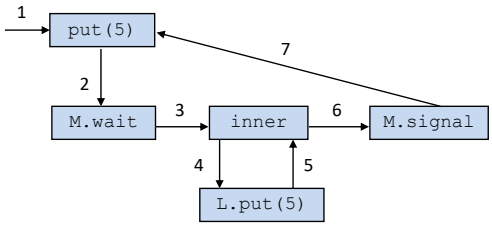

Figure 1. Monitor example and diagram showing the steps in execution of buffer.put(5)

also shown how to define a (singular) object[5] buffer, which is subclassed from Monitor.

The buffer object has a List object, L and two entry-methods, put and get. The statements of put and get are wrapped by the statements of entry, which ensures that they behave as critical regions. The steps of a method invocation buffer.put(5) are illustrated in Figure 1[6].

An object in Beta and qBeta may behave like a semi-coroutine in the style of SIMULA. An object may have executable statements like a method – called an *active object*. The execution of an object implies that its statements are executed. An object may execute a suspend-statement, which implies that control is returned to the point where it was executed. A subsequent execution of the object will resume execution after the point of suspension. A subsequent suspend executed by the object will again return to the caller, etc.

Figure 2 shows a simple example of a coroutine. The object main invokes do, which invokes S := foo, which invokes bar, which invokes go. Note that variable S is assigned a reference to the method invocation (an object) foo. The stage of execution at L1 is shown at the top part of the figure.

At the label L1, go executes this(foo).suspend, which implies that execution returns to the point after the invocation of foo. This is at the label L2 and illustrated by the bottom part of the figure. The execution of foo may later be resumed at L3 by execution of S.resume at L4. For a more detailed description of coroutines in Beta and qBeta, see [17, 20].

3 Simple concurrent processes

In [17] it is shown how to define safe concurrency frameworks for Monitor-like systems and Ada-like rendezvous-based systems. Here we show how to define a framework for

[4]Borrowed from [17, 20].

[5]A declaration of the form X: obj T{...} declares a singular object.

[6]Also from [17].

```
main: obj
   do():
      foo():
         bar():
            go():
               ...
               L1: this(foo).suspend
               L3: ...
            ...; go(); ...
         ...; bar(); ...;
      S: ref foo
      ...; S:= foo(); L2: ...;
      L4: S.resume; ...;
   ...; do(); ...
```

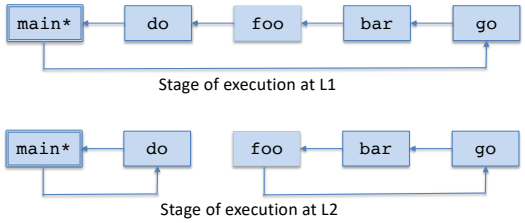

Figure 2. A simple coroutine

```
TempEx: obj LIB.SCP.System
   class Temperature: Process
      cycle
         controller.send(...measureTemp)
         sleep(500)
   temp1,temp2,temp3: obj Temperature
   controller: obj Process
      send(T: int): entry
         display.show(T)
         if T < minTemp then heater.turnUp(T)
         elif T > maxTemp then cooler.turnDown(T)
   display: obj Process ...
   heater: obj Proces
      turnUp(T: int): entry{...turnUpHeat(T)}
   cooler: obj Process
      turnDown(T: int): entry{...turnDownHeat(T)}
   ... start Process objects
```

Figure 3. Temperature system

Simple Concurrent Processes (abbreviated **SCP**). An **SCP** system consists of one or more concurrent processes. A process is an object subclassed from class `Process`. A `Process` may not access global data items, classes and methods, except for data items referring to `Process` objects.

A `Process` may define an interface consisting of one or more methods that must have `Entry` as supermethod – similar to the `Entry` method defined for the `Monitor` abstraction.

For a `Process`, `P`, other processes may execute invocations of the form `P.foo(...)`. The instance of `foo` is inserted into a local queue of `P` and `P.foo(...)` returns to the caller.

As mentioned, a `Process` may have statements. A `Process` may call the local method `exeNext` that will execute a possible `entry`-method in the queue. When a `Process` has finished executing its statements, it constantly executes possible methods

```
SCP: obj
   class System:
      class Process: ...
      sch: ref Scheduler
      SQS: obj ProcessQueue;
      inner;
      sch:= Scheduler()
```

Figure 4. The **SCP** framework

in the `entry`-queue. Methods in the queue are executed in the order of arrival.

The arguments of the `Entry`-methods must be either immutable objects or unique objects. A unique object may be referred to by at most one variable at any time during the program execution. Constraints ensure that an **SCP** system is safe in the sense that two or more processes cannot access the same data items at the same time. The constraints are enforced in the definition of the **SCP** framework using subpattern restrictions, but not shown here.

3.1 An example of an SCP system

In Figure 3, we show an example of an **SCP** system in the form of a simple system measuring the temperatures of an environment. The last temperature is shown on a display; if a temperature is below a certain boundary, a heater is activated and, similarly, if the temperature is above some limit, a cooler is activated. The system runs forever.

In `qBeta` modules are objects. The `TempEx` is such an object module. It is subclassed from `System` defined in `LIB.SCP`, which is the module defining the **SCP** framework. Three processes `temp1`, `temp2` and `temp3` being instances of class `Temperature` each measures the temperature at some point in the environment. The `control` object receives the temperatures from the `Temperature` processes. It sends the the temperature to the `display` and if below/above some boundary the `cooler` or `heater` is activated. Code prefixed by `'...'` as in `'...measureTemp'` is pseudo code that is not specified.

3.2 The SCP framework

The overall structure of the **SCP** framework is shown in Figure 4. The class `System` describes the outermost object enclosing an **SCP** system. Class `System` defines class `Process`, the scheduler `sch` and `SQS`, which is a FIFO-queue of active `Process` objects scheduled for execution.

The definition of the `Process` object is shown in Figure 5. A `Process` has a `start` method, which initializes the status of the `Process` to `ACTIVE` and inserts the `Process` into the queue of processes (`SQS`) scheduled for execution.

A `Process` has a super method `entry`, which must be the supermethod of all public methods of a subclass of `Process`. When executed, the actual submethod instance of `entry` is

```
class Process:
  start():
     status := ACTIVE
     SQS.insert(this(Process))
  entry():
     L.get()
     Q.insert(this entry)
     L.free()
     this(entry).suspend
     inner entry
  stop():< entry
     inner stop
     status:= TERMINATED
     this(Process).suspend
  exeNext():
     E: ref entry
     L.get()
     if Q.isEmpty then
        L.free()
        this(Process).suspend
        restart exeNext
     E := Q.removeNext()
     L.free()
     E.resume()
  status: integer; Q: obj Queue; L: obj Lock
  inner Process
  cycle{ exeNext() }
```

Figure 5. Class Process

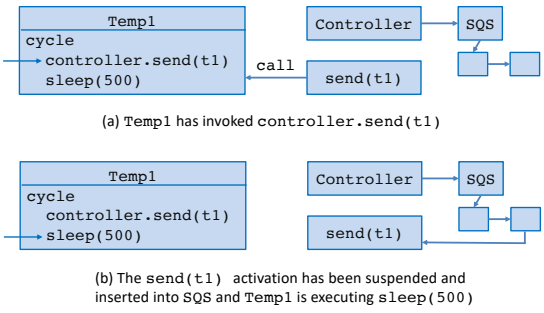

Figure 6. Illustration of a method suspend

inserted into the queue Q. Then the entry instance is suspended and control returns to the calling Process. This is illustrated in Figure 6.

The ability to detach a method invocation like an entry instance is a unique property of qBeta. In SIMULA, Beta and most other languages, it must be specified whether or not an instance of a class is a coroutine or just a plain object when the instance is generated. In qBeta, every object is potentially a coroutine.

Note that Q is a critical region – two or more processes may execute an entry method at the same time. Q is therefore, protected by a Lock L.

When a Process returns after inner, it constantly executes exeNext, which checks if there is an entry method in Q. If Q is not empty, then the next method is removed from Q and executed. The call E.resume(), resumes execution of E at the point of suspension – i.e. inner entry is executed and thereby the main part of the submethod of entry.

```
class Scheduler:
  active: ref Process
  cycle
     active := SQS.next()
     if active <> none then
        active.attach(100)
        if active.status = ACTIVE then
           SQS.insert(active)
```

Figure 7. Class Scheduler

If Q is empty, then the Process suspends execution and control returns to the scheduler – details will be shown in the next section. The Process will then be scheduled for execution and when resumed, it will restart execution of exeNext and test if an entry method has been inserted into Q. Again, access to Q is protected by the Lock L.

When a System object returns after inner, it generates and executes an instance of the Scheduler, which implies that scheduling is started.

3.3 Preemptive coroutines

In qBeta, a coroutine may be preempted. If S is an object, then S may be resumed by the statement S.attach(100) that is similar to S.resume, except that execution of S is preemptively suspended after 100 units of execution.

In Figure 7, we show how a Scheduler for the SCP may be implemented for a single-core processor. The Scheduler loops as long as there are Processes in SQS scheduled for execution – cycle ... repeatedly executes its do-part:

1. The next element in SQS is obtained by execution of active := SQS.next().
2. If active is not none, then active is resumed by active.attach(100).
3. After 100 execution units, active is preemptively suspended and control returns to the Scheduler.
4. If status of active is ACTIVE, active is inserted into SQS to be rescheduling later.
5. If active is none, then SQS is empty, Scheduler terminates, implying that the whole System terminates.

In the current implementation, the interpreter counts the number of bytecodes being executed, and when the appropriate number have been executed, the coroutine is suspended. For S.attach(100), S will be suspended when 100 byte codes have been executed. As an alternative, preemption points may be limited to allocation points or backward branches. Finally for some platforms, it may be possible to use a timer to trigger preemption.

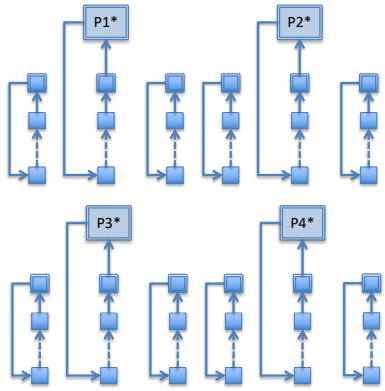

Figure 8. Snapshot of coroutines on multi-core platform

```
class Core:
  %core
  main: ref Object -- any object may be attached
  attach(P: ref Object): { main := P }
  Loop:
    if main = none then
       sleep(100)
       restart Loop
    else
       main
```

Figure 9. Description of a core class

4 Multi-core

Here we show how to implement preemptive coroutines on a multi-core platform. For this paper, we assume a multi-core platform with two or more cores and a shared memory.

The basic idea is to associate an active qBeta object with each core of a given platform. Such an active object may then execute other objects/coroutines like a scheduler in the style of the one shown above. There will thus be a number of truly concurrent schedulers, one for each core. Together these schedulers may handle the scheduling of Process objects. Figure 8 illustrates a situation with four cores and a number of Process objects to be scheduled.

The example in Figure 9 shows a class defining the structure of an active object that may be associated with a core. The property %core describes that an instance of class Core[7] may be associated with a core. Technically this means that if C is an instance of Core then C may be passed as an argument of a primitive operation fork(C). The actual semantics of fork(C) is implementation dependent. For implementations on a bare bone platform, C is associated a physical core – for implementations on top of an operating system like Windows and Linux, a native thread executing C is spawned.

The statements of the associated Core object are executed – and as long as main is none, the Core object sleeps for a

[7]The name of the class need not to be Core.

```
C1,C2,C3,C4: obj Core;
init_cores():
   fork(C1)
   C1.attach(new Scheduler())
   -- same for C2, C3 and C4
```

Figure 10. Binding objects to cores

```
class Scheduler:
   active: ref Process
    ...
SQS: obj -- critical region
   Q: obj ProcessQueue
   L: obj Lock
   insert(P: ref Process):
      {L.get(); Q.insert(P); L.free() }
  ...
```

Figure 11. SQS as a critical region

while and restarts the Loop. When a Scheduler is assigned to main – using attach – the Scheduler is executed.

For a platform with four cores, we may thus typically declare four instances of Core as shown in Figure 10. The situation is then that four Scheduler objects - like the ones described above – are executed simultaneously. For a concurrent program executed using these schedulers, this means that up to four coroutines may execute truly concurrent.

We need to make some changes to the Scheduler as described above in relation to SQS – the ProcessQueue – keeping track of scheduled Processes. In the uni-core version only one Scheduler object may access SQS. In the multi-core version SQS has to be a critical region since it may be accessed by two or more Schedulers. In Figure 11, SQS is defined as a critical region protected by a Lock, L.

In [17], the primitives disable and enable are used to disable and enable preemption of coroutines in order to implement critical regions. For a multi-core platform, these are not sufficient. Enable and disable enables and disables preemption within a single Scheduler – i.e. if disable is executed by an object, preemption is only disabled for the scheduler executing the object. The other schedulers are not disabled.

To handle critical regions in the multi-core situation, qBeta has a primitive operation, cmpAndSwap, equivalent to the hardware instructions found in many CPUs. The execution of cmpAndSwap ensures exclusive update of an integer variable. We may use cmpAndSwap to implement a Lock-class as shown in the lower right part of Figure 12. The get-method spins until M.cmpAndSwap(0,1) is successful, and the free-method assigns 0 (zero) to M and releases the lock.

In general, Lock is used for controlling critical regions. In [17], a Semaphore is defined using disable and enable of preemptive scheduling. And in turn the Semaphore is used to define a Monitor class and an Ada-like rendezvous class.

```
class Lock:
   M: var int; -- initially 0
   get():
      if M.cmpAndSwap(0,1) = 1 then restart get
   free(): { M := 0 }
```

Figure 12. Class Lock

The above scheduler for **SCP** is simpler than the ones for monitor and Ada-like rendezvous. For these systems, processes may be suspended waiting for a semaphore and the scheduler must keep track of possible waiting processes.

4.1 An alternative scheduler

The scheduler described above is probably too inefficient for many types of systems. It is, however, possible to define whatever scheduler is needed for at given task.

For the temperature example, if `temp1` is critical, we may then attach `temp1` to core1 (`C1`) as the only process. We may attach a scheduler to `C2` and let `temp2` and `temp3` be scheduled by this scheduler. And similarly attach a scheduler to `C3` handling `controller` and `display` and a scheduler attached to `C4` handling `heater` and `cooler`. The programmer has the full freedom to decide on the best setup.

5 Related work

Coroutines were originally proposed by Conway [6], and SIMULA was one of the first languages to support coroutines. Other examples include Modula 2 [25] and Icon [11]. In [9], Dahl and Wang presented a model of coroutines. The Lund SIMULA System introduced preemptive coroutines and was the direct inspiration for preemptive coroutines in Beta.

Beta generalized SIMULA style coroutines. The coroutine mechanism described in this paper is called *semi-coroutine*. SIMULA also supports *symmetric coroutines* where a coroutine explicitly transfers control to the next coroutine using a resume statement. Symmetric coroutines are the most common form of coroutines. In [19, 20] it is shown how to define a symmetric coroutine abstraction based on semi-coroutines.

Recently coroutines have been added to Python, Lua, Go, Kotlin and C++. In most of theses languages coroutines are similar to symmetric coroutines in SIMULA with cooperative scheduling and no support for preemptiveness and/or generators as in Icon.

In all of the above languages (except SIMULA and Beta) coroutines and tasks are different mechanisms. For Beta, coroutine is the basic mechanism for defining higher-level concurrency abstractions. The task mechanism as found in Java and similar OO languages is a built-in mechanism for supporting concurrency. Cooperative and preemptive coroutines may replace tasks and provide a much wider range of possibilities for defining concurrency abstractions.

Although concurrency abstractions are central for this paper, it is not in the scope of this paper to compare, evaluate and propose concurrency abstractions. This paper is neither about scheduling policies, load balancing or effective communication. The scheduler presented is only intended to illustrate the basic mechanisms for defining schedulers. In fact, the use of one global queue (SQS) for scheduled processes will be a bottleneck for programs consisting of many concurrent processes. Here a more fancier scheduling strategy will be needed. In principle, it is possible to implement any scheduling strategy in `qBeta`.

6 Status, discussion and further work

Coroutines have been used in SIMULA and Beta since the early days of object-oriented programming and this includes using preemptive coroutines for defining high-level concurrency frameworks. As mentioned, the disadvantage of such frameworks is that they are not safe with respect to race conditions. With subpatterns restrictions, it is possible to define safe concurrency frameworks. It is of course also possible to define frameworks that makes it possible to define objects that can be accessed without synchronization.

The basic synchronization primitive in Beta is a semaphore, which is often too complex and for `qBeta`, we have replaced it by a `cmpAndSwap`. Also Beta has only been implemented on a single-core platform.

`qBeta` is implemented on Ubuntu and Windows. In lack of direct access to control the cores, we simulate a core by a native thread. If we assume a platform with 4 cores, we allocate 4 threads and attach a scheduler to each of these threads. The operating system then schedules the native threads and `qBeta` schedules coroutines on top of these threads.

An experimental implementation of `qBeta` has been made for an ARM.[8] The compiler generates a boot image that can be run on a bare bone RPI. The implementation supports attaching objects to the different cores and executing them. We currently work on implementing synchronization. The primitive `cmpAndSwap` is implemented using exclusive read and write. We currently work on an implementation for ESP32 on top of FreeRTOS.

The experience with using `qBeta` is limited so far, but we have a proof-of-concept that preemptive coroutines can be used to define concurrent processes with full control over scheduling. We work on larger examples as well as support for non-blocking IO and support for handling interrupts.

Work is also needed on defining the right concurrency frameworks. The SCP example and the examples in [17] may cover one class of applications whereas alternatives may be needed for other kinds of applications. This same is the case for scheduling algorithms – different types of applications may need different kind of scheduling algorithms. But these are general issues independent of the work presented here.

[8] Raspberry PI 3 model B, Quadcore Cortex A 53.

Acknowledgment. The author would like to thank Birger Møller-Pedersen for fruitful discussions through many years and the anonymous referees for useful comments.

References

[1] Ada. Ada reference manual. proposed standard document, 1980.

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[3] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang.* Prentice Hall, 1993.

[4] Per Brinch-Hansen. The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, SE-1(2), 1975.

[5] Per Brinch-Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–35, 1999.

[6] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[7] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Simula 67 common base language (editions 1968, 1970, 1972, 1984). Technical report, Norwegian Computing Center, 1968.

[8] Ole-Johan Dahl and Kristen Nygaard. The development of the simula languages. In *ACM SIGPLAN History of Programming Languages Conference*, pages 439–480, 1978.

[9] Ole-Johan Dahl and Arne Wang. Coroutine sequencing in a block structured environment. *BIT*, 11:425–449, 1971.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java (TM) Language Specification.* Addison-Wesley, 1996.

[11] R.E. Griswold, D.R. Hanson, and Korb. J.T. Generators in icon. *ACM Trans. on Programming Languages and Systems*, 3(2):144–61, 1981.

[12] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language.* Addison-Wesley, 2003.

[13] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI'73 – 3rd international joint conference on Artificial intelligence*, pages 235–245, 1973.

[14] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

[15] B.W. Kernighan and D.M. Ritchie. *The C Programming Language 2nd edn.* Prentice Hall, Englewood Cliffs N.J., 1978.

[16] Bent Bruun Kristensen, Ole Lehrmann Madsen, and Birger Møller-Pedersen. The when, why and why not of the beta programming language. In Brent Hailpern and Barbara G. Ryder, editors, *History of Progamming Languages III*, pages 10–1–10–57. SIGPLAN, 2007.

[17] Ole Lehrmann Madsen. Building safe concurrency abstractions. In G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K Taura, editors, *Concurrent Objects and Beyond*, pages 66–104. Springer, 2014.

[18] Ole Lehrmann Madsen. Using coroutines for multi-core preemptive scheduling — extended version. Technical report, Department of Computer Science, Aarhus University, 2021. This paper is an extended version of the paper published at the 11th Workshop on Programming Languages and Operating Systems (PLOS 2021). It may be be obtained from cs.au.dk/ olmadsen.

[19] Ole Lehrmann Madsen and Birger Møller-Pedersen. What object-oriented programming may be—and what it does not have to be. In S. Gjessing and K. Nygaard, editors, *ECOOP'88 – European Conference on Object-Oriented Programming*, volume 322 of *Lecture Notes in Computer Science*, pages 1–20. Springer Verlag, 1988.

[20] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison Wesley, 1993.

[21] P. Naur. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6, 1963.

[22] python.org. The python tutorial, 2011.

[23] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading MA, 1986.

[24] J. Vaucher and B. Magnusson. Simula frameworks: the early years. In Mohammad Fayad and Ralph Johnsson, editors, *Object-Oriented Application Frameworks*, page 672. Wiley, 1999.

[25] N. Wirth. *Programming in Modula-2.* Springer-Verlag, Heidelberg, New York, 1982.

[26] Yasuhiko Yokote and Mario Tokoro. Experience and evolution of concurrent smalltalk. In *OOPSLA '87– Object-Oriented Programming Systems, Languages and Applications*, pages 406 – 415. ACM SIGPLAN, 1987.

[27] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in abcl/1. In *OOPLSA '86 – Object-Oriented Programming Systems, Languages and Applications*, pages 258 – 268. ACM SIGPLAN, 1986.

A A short introduction to qBeta

In this appendix, we describe some further details of qBeta as used in this paper.[9] As mentioned in the introduction, qBeta is a further development of xBeta as used in [17] and xBeta is based on Beta. For the purpose of this paper, the differences between qBeta and xBeta are not important. The main differences between Beta and qBeta/xBeta are with respect to syntax, coroutines, synchronization primitives and the possibility of binding objects to physical cores of the underlying platform.

In qBeta, nesting of classes and methods (block-structure) is defined by indentation as in Python, but it is also possible to use curly brackets({...}) to describe block-structure and semicolon (;) to separate declarations and/or statements. For the benefit of the reader, we will use a syntax in the style of C++, Java, and C#, to avoid introducing qBeta syntax for classes, methods, statements and expressions.

We refer to Beta below for elements of qBeta that are similar to Beta and qBeta for elements that differ from Beta.

In Beta there is no distinction between a class and a method – they are unified into the notion of a *pattern*. A pattern may be used as a class or method and instances of a pattern may be executed as method activations, or as coroutines.

A *class* in the following means a pattern intended to be used as a class. A similar terminology is used for *method*, *coroutine*, etc.

A class (pattern) has the following syntax:

```
class MyClass: super
   attributes
   statements
```

Where super is the superclass of MyClass, attributes describes the attributes of the objects and statements describes a sequence of statements.

As can be seen, a class pattern may contain a action part in the form of statements to be executed. A method pattern has the following syntax:

```
returnType msg(arguments): super
   attributes
   statements
```

The structure of a method is the same as the structure of a class with respect to super, attributes and statement part.

A class may in fact also have arguments and a return type, but we will not give examples of this in this paper. And since there is no difference between a class and a method, the keyword **class** is only used for the benefit of the reader.

A distinctive feature of Beta is that a method pattern may inherit from another method – its *supermethod*. And methods may thus be organized in a method hierarchy similar to a class hierarchy.

Consider the following example of a submethod hierarchy:

```
void msg1():
   ...
   S1;
   inner msg1
   S6
void msg2(): msg1
   ...
   S2
   inner msg2
   S5
void msg3(): msg2
   ...
   S3
   inner msg3
   S4
```

The methods msg2 and msg3 are submethods of msg1 and msg2 respectively, and msg1 and msg2 are *supermethods* of msg2 and msg3 respectively. Execution of msg1 implies that S1, inner msg1 and then S6 are executed, and in this case, inner msg1 is just a skip-statement with no effect.

Execution of msg2 starts by execution of the action part of msg1, and in this case, execution of inner msg1 implies that the statement part of msg2 is executed. All together the following statements are executed: S1, inner msg1, S2, inner msg2, S5 and then S6. In this case, inner msg1 has an effect whereas inner msg2 is a skip-statement.

Submethods may form an arbitrary hierarchy – the method msg3 is a submethod of msg2, which in turn is a submethod of msg1.

Execution of msg3 gives rise to the following statements being executed:

1. S1,
2. inner msg1,
3. S2,
4. inner msg2,
5. S3,
6. inner msg3 , -- skip, empty action
7. S4,
8. S5,
9. S6.

In Beta, it is possible to define *singular objects* – corresponding to anonymous classes in e.g. Java. A singular object myObj may be declared as follows:

```
myObj: obj
   S: ref T
   foo:
      ...
```

myObj has a data item S that may refer to instance of class T, and a local pattern foo.

In a similar way, it is possible to define *singular method activations*. A singular method activation may have the form:

```
msg3
   ...
   Sx
```

[9] This appendix is a modified version of a similar description of Beta in [17].

Execution of this statement implies that the action part of the top supermethod, in this case msg1, is executed and the resulting statements being executed are: S1, inner msg1, S2, inner msg2, S3, inner msg3, Sx, S4, S5, S6. In this case inner msg3 is not the empty action, but implies execution of Sx.

A singular method activation corresponds to a prefixed block in SIMULA, which in turn is a generalization of inner blocks from Algol 60 [21].

In Beta, control abstractions are defined using submethods. A simple example is the control pattern cycle that may be used to execute a list of statements infinitely unless escaped by a break statement - see below.

```
cycle:
   inner cycle
   restart cycle
```

Pattern cycle may be used as follows:

```
cycle
   S1
   S2
   S3
```

The statements S1; S2; S3 are the executed forever unless a *break* statement is executed as part of S1, S2 or S3.

Beta has two break statements, restart L as used in cycle, and leave L. They both have the same effect as a goto statement, but can only be used in a pattern with the name L or a structured statement labelled by L.

Submethods may also be used to define iterators as shown in the example below of a List class:

```
List:
   void insert(V: integer): {...};
     integer head(): {...};
     void scan():
       current: integer
       current := first_element;
       while current <> none then
          inner scan;
          current := next_element;

     -- representation of List
   }
void useList():
  L: obj List;
  ...
  L.scan{ current.print()}
```

The List has methods insert and head (returns and removes the head of the list). In addition, List has a control pattern scan that iterates through all elements of the list – none is the the same as null in Java. For each element of the list, the variable current holds the value of the next element and inner scan is executed. Execution of inner scan implies that the statement part of a submethod of scan is executed. Note that inner scan appears within a loop and is thus potentially executed several times.

The method useList defines a List object L, it scans the elements of L, and prints each element.

In Beta, a declaration likeL: **obj** List defines a static object in the form of L being a constant reference that refer to an instance of List, which is generated as part of the generation of the object enclosing L. It is similar to final List L = new List() in Java. The keyword **obj** declares a constant reference wheres the keyword **ref** declares a variable reference. In S: **ref** List, S may refer to different instances of S during execution.

Execution of the anonymous method activation L.scan{current.print()} thus prints the elements of L. Note that L.scan is the supermethod of the anonymous method activation.

Control abstractions defined using submethods and inner scan have the advantage that you do not need to initialize the iterator and there is no state in the object keeping track of the progress of the iterator. This means that several instances of the control abstraction (iterator) can be made.

Submethods are also useful for describing a general supermethod that ensures mutual access to shared variables. One example of this is the entry method defined in class Monitor in Figure 1 of Section 2. Another similar example is the entry method defined in class Process of Figure 5 in Section 3.2.

For further details about Beta, see [16, 20]. Reference [20] is a book about Beta and is available as a PDF-file from cs.au.dk/~olmadsen.

B Defining safe concurrency abstractions

In this appendix, we briefly show how to add subpattern restrictions to the SCP-framework in order to make it safe.

Per Brinch-Hansen has pointed out that a language like Java is inherently insecure with respect to race-conditions and this is also the case for most other class-based object-oriented languages like C++, C# and Beta. The synchronized mechanism in Java supports monitor-like behavior, but the compiler does not enforce the programmer to annotate a method as synchronized.

Beta has the same problem as Java since the compiler cannot check that all interface methods for a monitor as e.g. shown in Figure 1 in fact are submethods of entry.

For the SCP-framework defined in Figure 4, we described a number of restrictions that the framework should obey, but these cannot be checked by the compiler. As mentioned, we have introduced the notion of subpattern restrictions [17] to be able to define the necessary restrictions to be enforced on the use of a framework like SCP.

In the following, we show how subpattern restrictions may be used to enforce the necessary restrictions on the SCP-framework:

An object being a subclass of Process should not be able to modify global variables except that it may invoke methods

```
SCP: obj
   class System:
      class Process:
         %globals Process,SQS
         %interface entry
         entry:
            %arguments value,immutable,unique
         ...
      sch: ref Scheduler
      SQS: obj ProcessQueue;
      inner System;
      sch:= Scheduler()
```

Figure 13. The **SCP** framework

in other Process objects, and add itself to the SQS queue. This may be enforced by means of the %globals property:

```
%globals P1, P2, ..., Pn
```

A pattern (class or method) annotated by the %globals-property may only access global data-items of type P1, P2, ... Pn.

All methods in the public interface of a Process object must inherit from pattern entry. This can be enforced by the %interface property:

```
%interface P1,P2,..,Pn
```

For a pattern annotated by the %interface property, the public interface must be methods that inherit from one of P1, P2,.., or Pn.

In order to prevent race conditions, we require that the arguments of a public method of class Process must be values, references to immutable objects or references to unique objects.

In qBeta, it is possible to define value objects that differ from ordinary objects in the sense that assignment of an object Y to X (X := Y) means that data-items of Y are assigned to the corresponding data-items of X. In a similar way, a comparison of X and Y (like X = Y) consist of a comparison of similar data-items in X and Y. This is in contrast to reference assignment and reference comparison for traditional objects.

In qBeta, all basic patterns like, integer, Boolean, char and float define value objects.

In qBeta, it is also possible to declare an object to be immutable by means of the %immutable property:

```
%immutable
```

The state of an immutable object cannot be changes after it has been generated and initialized.

A unique object has at most one reference at a given time during execution:

```
%unique
```

The complete definition of the SCP-framework with subpattern restrictions is shown in Figure 13.

The Process class has been annotated with %globals Process,SQS, which implies that the only global objects that can be accessed in objects being subclassed from Process are other Process objects of the SQS object.

In addition, the and %interface entry property enforce the property that all public methods of subclasses of Process must inherit from and entry.

For the method pattern entry, the property %arguments value,immutable,unique, ensure that the arguments of a submethod of entry must be value objects, immutable objects or unique objects. For the temperature system in figure 3, the arguments are all basic integers and thus value objects.

All the subpattern restrictions may be expressed in terms of predicates over a number of domains of the program execution. We do not use a predicate notation since the compiler will in general not be able to validate such a predicate at compile time. The predicates being used here may thus be considered names or abbreviations of specific predicates.