SCOOP: Concurrent Programming Made Easy

Piotr Nienaltowski	Volkan Arslan	Bertrand Meyer
Chair of Software Engineering	Chair of Software Engineering	Chair of Software Engineering
ETH Zurich	ETH Zurich	ETH Zurich
8092 Zurich	8092 Zurich	8092 Zurich
Switzerland	Switzerland	Switzerland
+41 16 32 44 68	+41 16 32 44 70	+41 16 32 04 10
Piotr.Nienaltowski@inf.ethz.ch	Volkan.Arslan@inf.ethz.ch	Bertrand.Meyer@inf.ethz.ch

ABSTRACT

The **SCOOP** model (Simple Concurrent Object-Oriented Programming) provides a simple yet very powerful mechanism for parallel computation. The model takes advantage of the inherent concurrency implicit in object-oriented programming to provide programmers with a simple extension enabling them to produce parallel applications with little more effort than sequential ones. SCOOP is applicable to many different physical setups, from multiprocessing to multithreading, highly parallel processors for scientific computation, distributed computation, and Web services. In this article, we present the basic concepts of the model and show how easily parallel and distributed computation can be achieved in a way that preserves the full power of all object-oriented techniques. Several programming examples illustrate the discussion.

Categories and Subject Descriptors

Distributed, mobile, parallel, and real-time systems.

General Terms

Languages, design.

Keywords

Concurrent object-oriented programming, parallel and distributed computation, Eiffel, Design by Contract™.

1. INTRODUCTION

In traditional sequential programming, the object-oriented model has gained wide acceptance. Concurrent and distributed programming remains, however, one of the last areas of software engineering where no single direction has been generally recognised as the preferred approach. Frameworks such as CSP, CCS, and π-calculi enjoy strong academic support, but they remain far from the techniques actually applied in the industry.

Finding a satisfactory framework for concurrent and distributed development is an urgent issue for the industry. Concurrent programming has become a required component of ever more types of application, including some that were traditionally thought of as sequential in nature. Beyond mere concurrency, today's systems have become distributed over networks, including the network of networks — the Internet. The telecommunication industry is in particularly dire need of simple, teachable techniques, directly supported by tools, which can guarantee the efficient production of correct and robust software providing a high Quality of Service.

The SCOOP model offers a comprehensive approach to building high-quality concurrent and distributed systems. The idea of SCOOP is to take object-oriented programming as given, in a simple and pure form based on the concepts of Design by Contract [8], which have proved highly successful in improving the quality of sequential programs, and extend them in a minimal way to cover concurrency and distribution. The extension indeed consists of just one keyword `separate`; the rest of the mechanism largely derives from examining the consequences of the notion of contract in a non-sequential setting. The model is applicable to many different physical setups, from multiprocessing to multithreading, network programming, Web services, highly parallel processors for scientific computation, and distributed computation. Writing applications with SCOOP is extremely simple, since programmers do not need to deal with low-level concepts typically used in concurrent programming (semaphores, rendezvous, monitors etc.).

The SCOOP model has undergone several refinements since its introduction in [10]. Currently, there is no publication taking into account these developments and describing the model in its entirety. This article tries to fill this void by presenting an up-to-date, cohesive view of SCOOP as it is today.

The rest of the article is organised as follows: Section 2 presents the SCOOP model; its basic concepts are described. Section 3 focuses on distributed programming in SCOOP. Section 4 presents library mechanisms of SCOOP. Section 5 discusses the design criteria of the model; it also presents related work by other authors. Finally, Section 6 summarises the article and points out our current and future research directions.

2. THE SCOOP MODEL

SCOOP stands for *Simple Concurrent Object-Oriented Programming*. Indeed, the very power of the model lies in its simplicity. More precisely, the extension covering full-fledged concurrency and distribution is as minimal as it can get starting from a sequential notation: SCOOP adds a single new keyword to the Eiffel programming language — `separate`.

Before describing the model itself, let us review the essential criteria that should guide the development of an object-oriented mechanism for concurrent and distributed programming. These criteria have served as a basis for the model presented here (see 5.1 for a more in-depth discussion). The goals include:

- minimality of mechanism,
- full use of inheritance and other object-oriented techniques (e.g. dynamic binding, genericity),
- compatibility with Design by Contract,
- provability,
- applicability to many forms of concurrency,
- support for the reuse of non-concurrent software,
- efficient deadlock-avoidance scheme,
- adaptability through libraries.

2.1 Architecture

SCOOP has a two-level architecture (see Figure 1). The top layer of the mechanism is platform-independent. This is the layer which most applications use, and which this article describes. To perform concurrent computations, applications simply use the separate mechanism implemented at this level. Internally, the implementation relies on some practical concurrent architecture (bottom layer). Figure 1 lists some possibilities:

- an implementation using the .NET platform, especially its Remoting and Threading mechanisms [12],
- a thread-based implementation, e.g. with POSIX threads,
- a multi-threading implementation on a real-time operating system, e.g. Windows CE .NET with the .NET Compact Framework.

The proposed architecture largely simplifies the model: the general concurrency mechanism implemented in the top layer is very straightforward, and all the platform-specific features can be accessed via libraries (see section 4).

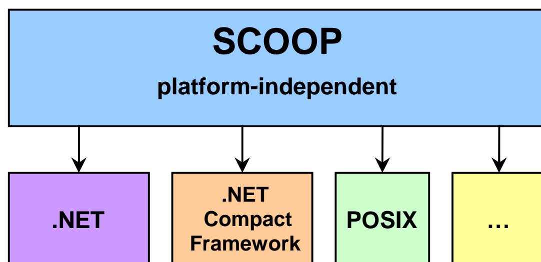

Figure 1. Two-level architecture of SCOOP

2.2 Processors

SCOOP uses the basic scheme of the object-oriented computation: the feature call, e.g. `x.f(a)`, which should be understood in the following way: the client object calls feature `f` on the supplier object attached to `x`, with argument `a` (see Figure 2). In a sequential setting, such calls are synchronous, i.e. the client is blocked until the supplier has terminated the execution of the feature. To introduce concurrency, SCOOP allows the use of more than one *processor* to handle execution of features. If different processors are used for handling the client and the supplier objects, the feature call becomes asynchronous: the computation on `o1` can move ahead without waiting for the call on `o2` to terminate (see Figure 2). Hence the asynchronous semantics of such feature calls.

Processors are the principal new concept for adding concurrency to the framework of sequential object-oriented computation. A concurrent system may have any number of processors, as opposed to just one for a sequential system. In the SCOOP model, a processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects. It should not be confused with a physical CPU. In fact, it can be implemented by a piece of hardware (a computer), but also by a process of the underlying operating system, or a single thread. In the .NET Framework, processors can be mapped to *application domains* [12]. Viewed by the software, a processor is an abstract concept; the same concurrent application may be executed on very different architectures without any change to its source text (see section 3).

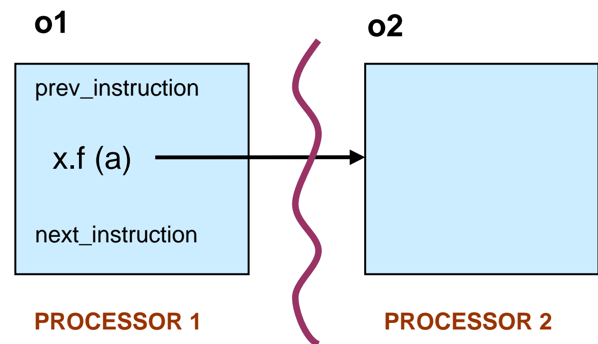

Figure 2. Feature call in SCOOP

2.3 Separate calls

Since the effect of a call depends on whether the client and the supplier objects are handled by the same processor or by different ones, the software text must indicate that fact unambiguously. A declaration of an entity or function, which normally appears as `x: SOME_CLASS` may now also be of the form `x: **separate** SOME_CLASS`. Keyword `separate` indicates that entity `x` is handled by a different processor, so that calls on `x` should be asynchronous and can proceed in parallel with the rest of computation. With such a declaration, any creation instruction `create x.make (…)` will spawn off a new processor to handle calls on `x`. We do not specify which processor to use for handling the object. The important thing is the fact that this processor is different from the processor handling the current object[1].

Instead of declaring a single entity `x` as separate, the declaration of its base class may also be of a new form: `**separate class** SOME_CLASS`. In this case `SOME_CLASS` will be called a *separate class*[2], and all its instances will be *separate objects*. The following conventions follow:

- a type is separate if:

---

[1] In section 3.1 we describe how processors are mapped to physical resources.

[2] It follows from the Eiffel syntax convention that a class may be at most one of: *separate*, *expanded*, or *deferred* [8]. The separateness of a class is not inherited: a class is separate or not according to its own declaration, regardless of its parents' status.

- it is based on a separate class, or
- it is of the form **separate** T for some T (T itself may be non-separate or separate),

- an entity is separate if its type is separate,
- an object is separate if it is attached to a separate entity,
- a function is separate if its type is separate,
- an expression is separate if it is either a separate entity or a call to a separate function,
- a call or creation instruction is separate if its target is a separate expression,
- a precondition clause is separate if it involves a separate call.

If a target of a call is a separate expression, i.e. a separate entity or an expression involving at least one separate entity, such call is referred to as *separate call*. In Figure 2, x is a separate entity, o2 is a separate object, and x.f(a) is a separate call.

2.3.1 Validity of separate calls

The validity of separate calls is governed by the *Separateness Consistency Rules*:

- If the source of an attachment (assignment instruction or assignment passing) is separate, its target entity must be separate too.
- If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate.
- If the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate.
- If an actual argument of a separate call is of an expanded type[3], its base class may not include, directly or indirectly, any non-separate attribute of a reference type.

2.4 Contracts in a concurrent setting

As mentioned before, SCOOP relies on the principles of Design by Contract. Therefore, before discussing the access control policy for SCOOP, let us have a closer look at the relation between contracts and concurrent execution.

2.4.1 Preconditions

The semantics of preconditions is different in sequential and concurrent setting. In sequential programs, preconditions are assertions that have to be fulfilled by the client object before calling the routine of the supplier object. If one or more preconditions are not met, the contract is broken and an exception is raised in the client object. The contract is broken because the client has not fulfilled the requirements *before calling* the given routine. For instance, it has tried to store a value into a full buffer. Since the execution is sequential, the state of the buffer cannot change (no other client can try to consume an element from the buffer in the meantime) so the only solution is to signal the abnormal situation by raising an exception.

Let us examine a similar situation in a concurrent context (see Example 1). The buffer may be full *when* the client object is trying to store a value into it, but nothing prevents another client object from consuming an element from the buffer *later on*. Therefore the buffer may become non-full at some point of execution, and the client object attempting to store an element can succeed. A non-satisfied precondition does not break the contract, it just forces the client object to wait until the precondition is satisfied. This simple example shows that preconditions not involving any separate entities (e.g. value_provided) keep their original semantics also in a concurrent setting (they are *correctness conditions*); on the other hand, preconditions involving separate expressions (e.g. buffer_not_full) become *wait conditions*.

[3] Entities of an *expanded type* represent directly an object, not a reference to it [8]. Examples: INTEGER, REAL, BOOLEAN.

Example 1. Preconditions and postconditions

```
store (buffer: separate BUFFER [INTEGER];
       value: INTEGER) is
    -- Store value into buffer.
  require
    buffer_not_full: not buffer.is_full
    value_provided: value /= Void
  do
    buffer.put (value)
  ensure
    buffer_not_empty: not buffer.is_empty
  end
```

2.4.2 Postconditions

The consequences of concurrent execution on the properties of a call are twofold. On one hand, to satisfy the contract, the client has fewer properties to ensure before the call (only the preconditions clauses that do not involve separate entities). On the other hand, the client should be more careful when relying on postconditions that involve separate entities. In Example 1, the postcondition buffer_not_empty is satisfied when the call is finished, but the client cannot rely on it (e.g. use it in the if…then statement) within the scope of the feature (say r) that contains the call to store unless the actual argument passed to store is a formal argument of r. Outside that scope, the postcondition may not hold, since other clients may have invalidated it in the meantime.

2.4.3 Invariants

Another interesting problem is the relation between concurrent execution and *class invariants*. The class invariant is the most important part of a contract because it ensures the consistency of the class instances (objects). An object-oriented software system can be consistent if and only if every object in the system is consistent with its specification (its base class).

At first, the connection between class invariants and concurrency does not seem so obvious. If one tries to reason about concurrent programs written in SCOOP, the use of invariants is indeed exactly the same as in a sequential context (see section 2.7). This is because the access control policy of SCOOP allows only one feature to be called on the supplier object at any time (see section 2.5), so satisfying or violating the invariant of the supplier's base class depends only on the outcome of this routine call. Suppose that we try to change the locking policy and allow concurrent execution of several routines of the same supplier object. If two or more of these routines may change the invariant, we should make sure that these routines are not executed at the same time. Otherwise, the invariant may be violated, even if the sequential execution of the same routines would not violate it. On the other

hand, if we are sure that the routines do not change the invariant, we are allowed to execute them concurrently [11].

2.5 Access control policy

Controlling the access to shared resources is the main problem in concurrent computation. In non-object-oriented settings the concept of "critical section" is used: it is simply a code fragment in which a shared resource is accessed. At most one process[4] can be executing the critical section at any given time. Efficient solutions to conflict problems must be characterised by a synchronisation among processes, so that they have to wait for executing a critical section if another process is accessing the shared resource. This kind of synchronisation is called "mutual exclusion" (from running the critical section at the same time).

The situation changes significantly when we deal with object-oriented computations. Explicit critical sections are not required any more, since they may be encapsulated in class routines. The most important question is: how to ensure that concurrent calls to the routines of the same object do not cause deadlock, and do not violate the integrity of the object (i.e. the invariant of its base class)? An appropriate locking policy should be applied in order to ensure these two conditions.

SCOOP does not use the concept of critical section, instead it relies on the mechanism of argument passing. For a separate call to be valid, the target of the call must be a formal argument of the enclosing routine. Such "embedding" of separate calls in routines allows exclusive locking of separate objects.

Consider Example 2. We deal with the producer-consumer synchronisation. Assume that several producer objects are producing integer values and storing them into the shared buffer `buf`; several consumer objects are consuming elements from that buffer. From the point of view of both the producers and the consumers, `buf` is a separate object (that is why it is declared as `separate` in the source code of both classes). In order to perform any call to `buf`, a client object (be it producer or consumer) must obtain an exclusive lock on `buf`. Since SCOOP relies on the argument passing mechanism for this purpose, the target of a separate call must appear as an argument of the enclosing routine; that is why all the calls to `buf` are embedded into routines `store` and `consume_one`. Direct calls to `buf.put`, `buf.item`, and `buf.remove` are forbidden (see the example).

Example 2. Producer-consumer synchronisation[5]

```
class PRODUCER

feature

 store (buffer: separate BUFFER [INTEGER];
        value: INTEGER) is
      -- Store value into buffer.
  require
    buffer_not_full: not buffer.is_full
    value_provided: value /= Void
  do
    buffer.put (value)
  end

 random_gen: RANDOM_GENERATOR

 buf: separate BUFFER [INTEGER]

 produce_n (n: INTEGER) is
      -- Produce n integer values and store
      -- them into a buffer.
   local
     value: INTEGER
     i: INTEGER
   do
     from i := 1
     until i > n
     loop
       value := random_gen.next
       store (buf, value)
         -- buf.put (value) is forbidden
         -- here
       i := i + 1
     end
   end

end -- class PRODUCER


class CONSUMER

feature

 consume_one (buffer: separate
               BUFFER [INTEGER])
   is
     -- Consume one element from buffer.
   require
     buffer_specified: buffer /= Void
     buffer_not_empty: not buffer.is_empty
   do
     value := buffer.item
     buffer.remove
   end

 buf: separate BUFFER [INTEGER]

 consume_n (n: INTEGER) is
      -- Consume n elements from a buffer.
   local
     i: INTEGER
   do
     from i := 1
     until i > n
     loop
       consume_one (buf)
         -- buf.item and buf.remove are
         -- forbidden here
       i := i + 1
     end
   end

end -- class CONSUMER
```

Let us have a closer look at the locking mechanism. When a consumer object is making a call to `consume_one` inside routine `consume_n`, it passes `buf` as argument to that call. According to the SCOOP access control policy, when one or more arguments of a routine are separate objects, the client must obtain exclusive locks on all these objects before executing the routine.

[4] Here process = thread of execution. It may be called process, thread, processor, etc.

[5] To simplify the example, the postconditions have been omitted.

Therefore, the consumer object in our example must obtain an exclusive lock on `buf` before executing `consume_one`. If another object is currently holding the lock, the client has to wait until the lock has been released, and then try to acquire it. When the client has finally acquired the lock, the preconditions are checked. If all the preconditions hold, the routine is executed, and the lock is released after the routine has terminated the execution. Should one or more preconditions involving separate objects (i.e. wait conditions, see 2.4.1) not hold, the client releases all the locks and restarts the whole process from the beginning: first acquiring the locks, then checking the preconditions[6]. This allows other clients to access the supplier object, hopefully changing its state, so that the wait conditions required by our client are eventually met.

The locking policy of SCOOP states that at most one client may access any supplier object at any given time. This ensures that (correct) separate calls do not violate the integrity of the supplier object. It also makes it easier to reason about concurrent programs. Since only one client object can hold a lock on the supplier object at any time, interference between several client objects is impossible. Therefore, one can easily decide which object is responsible for possible breaches in the contract (e.g. breaking the invariant of the class corresponding to the supplier object).

2.5.1 Scheduling policy

The scheduling policy in SCOOP ensures that a separate call is scheduled as soon as all the necessary locks can have been acquired on the supplier object(s) and all the wait conditions are satisfied. The calls on the same supplier are executed in the FIFO order, and there is no starvation. The implementation of such policy is straightforward (it has been done, for instance, in the SCOOPLI library [12]).

2.6 Synchronisation and wait by necessity

Thanks to the asynchronous semantics of separate calls, clients executing such calls are not blocked and can proceed with the rest of their computation. But surely a client may need to resynchronise with the supplier. When should we wait for the call to terminate?

It would seem that a special mechanism is needed, as has been proposed by some concurrent object-oriented languages such as Hybrid, to reunite the client computation with the separate call that has been made. In SCOOP, no explicit wait mechanism is needed: instead, we use the idea of *wait by necessity*, introduced by Denis Caromel [4]. The goal is to wait only when we truly need to, but no earlier.

When does the client need to make sure that a call `x.f (...)`, for `x` attached to a separate object `o1`, is finished? Not when the client is doing something else on other objects, be they separate or not; not even necessarily when it has started a new procedure call `x.g (...)` on the same separate object (in such situations, subsequent calls can be simply logged so that they can be processed in the FIFO order). The client should wait if and only if it needs to access some property of `o1`. *Then* `o1` must be available, and all preceding calls to it must have finished.

According to the Command-Query Separation principle [8], features of a class can be divided into commands (procedures), which perform some transformation on the target objects, and queries (functions and attributes) which return information about it. Command calls do not need to wait, but query calls may.

Example 3. Wait by necessity with a single supplier

```
consume_two(buffer: separate BUFFER) is
     -- Consume two elements from buffer.
   local
     value: INTEGER
   require
     buffer_specified: buffer /= Void
     at_least_two_elements:
              buffer.count >= 2
   do
     value := buffer.item
     buffer.remove
     value := buffer.item
     buffer.remove
   end
```

Consider the feature `consume_two` in the example above. Using this feature, a consumer object consumes two subsequent elements from a buffer. The first assignment `value := buffer.item` can be executed without waiting. After calling `buffer.remove`, but before the call has finished, the client tries to call `buffer.item` once again. Since `buffer.item` is a query call, the client cannot proceed: it *must wait* for the previous call (`buffer.remove`) to terminate.

Wait by necessity also applies to situations where calls to several separate objects are involved. Consider the following example:

Example 4. Wait by necessity with multiple suppliers

```
some_feature (x,y,z: separate SOME_CLASS) is
     -- Make some calls to x, y, and z
  do
    x.f
    y.f
    x.g
    z.f
    y.g
    v := x.is_empty
    v := x.value > y.value
  end
```

The client object makes subsequent calls to features of the separate supplier objects `x`, `y`, and `z`. The call `y.f` can proceed without waiting for `x.f` to terminate. Also the calls `z.f` and `y.g` do not need to wait, neither for `x.f` nor for `x.g`. Still, the call `y.g` has to wait for `y.f` to finish, because they involve the same separate object. Similarly, `x.g` has to wait until `x.f` has terminated. The first assignment involves objects `z` and `x`, therefore it is blocked until all previous calls on these objects have terminated. On the other hand, it does not involve object `y`, so it can be executed even before `y.f` and `y.g` have terminated. The second assignment involves all three separate objects `x`, `y`, and `z`. Therefore, its execution will be blocked until all the

[6] We only consider wait conditions here. As already explained in section 2.4.1, preconditions that do not involve any separate entities are correctness conditions, so their violation is handled in the same way as in a sequential setting, i.e. by raising an exception in the client.

previous calls have terminated. Here are a few examples of a correct schedule[7] (run):

- `x.f ‖ y.f ; y.g ‖ x.g ‖ z.f ;`
  `v := x.is_empty ; v := x.value > y.value`
- `x.f ; z.f ‖ x.g ; v := x.is_empty ‖ y.f ;`
  `y.g ; v := x.value > y.value`
- `z.f ‖ y.f ‖ x.f ; x.g ; v := x.is_empty ‖`
  `y.g ; v := x.value > y.value`

These observations yield the basic concept of wait by necessity: if a client has started one or more calls on a certain separate object, and it executes on that object a call to a query, that call will only proceed after all the earlier ones have been completed, and any further client operations will wait for the query call to terminate.

2.6.1 An optimisation

We may go further by examining whether the query's result is of an expanded type or a reference type. If the type is expanded, for example if it is BOOLEAN or another of the basic types, there is no choice: the client needs the value, so it must wait until the query has computed the result. On the other hand, for a reference type, one can imagine that a smart implementation could still proceed while the result, a separate object, is being computed; in particular, if the implementation uses proxies for separate objects, the proxy object itself can be created immediately, so that the reference to it is available even if the proxy does not yet refer to the desired separate object.
Such optimisation, however, complicates the concurrency mechanism because it means that proxies must have a "ready or not" boolean attribute, and all operations on separate references must wait until the proxy is ready. It also seems to prescribe a particular implementation – through proxies. Therefore, we do not retain it as part of the basic SCOOP mechanism.

2.7 Proving the correctness of programs

After each feature call, the invariant of the object's base class should be satisfied. This is necessary for preserving the consistency of the object. Let us have a look at the lifecycle of an object (see Figure 3).

The object is externally observable only in the states represented by `S1`, `S2`, `S3`, etc. That is to say, after the creation of the object and after every application of a feature on that object. To prove the correctness of the underlying class, we only have to verify that the following properties hold:

Property 1. For a creation procedure `make`, if the body of `make` is executed when the object has been initialised to the default values and the precondition holds, the resulting state will satisfy the postcondition and the invariant. It can be expressed as:

$$\{\text{default } \mathbf{and} \text{ pre}_{make}\} \text{ body}_{make} \{\text{post}_{make} \mathbf{ and} \text{ INV}\}$$

[7] `a‖b` parallel execution (`a` and `b` overlap)
`a;b` sequential execution (`b` executes after `a` has terminated)
‖ binds stronger than `;`

Property 2. For an exported feature `f`, if the body of `f` is executed when the precondition and invariant hold, the resulting state will satisfy the postcondition and the invariant:

$$\{\text{pre}_f \mathbf{ and} \text{ INV}\} \text{ body}_f \{\text{post}_f \mathbf{ and} \text{ INV}\}$$

In the sequential context, there are no complicated run-time scenarios to analyse. Therefore, we can rely on such simple properties to check the consistency of objects, as well as the consistency of the whole software system. Introducing concurrent execution complicates things: we have to analyse all possible interleavings, which leads to a combinatorial explosion of cases to consider. For that reason, SCOOP lets at most one routine execute on any given object at a time. The result of such restriction is the *single locking policy*[8].

Certainly, the two properties mentioned above are not sufficient to prove the correctness of a software system. Nevertheless, they constitute a first step in the process of devising a mathematical model for reasoning about concurrent programs. We are convinced that such model will be very helpful in proving the correctness of SCOOP-based programs.

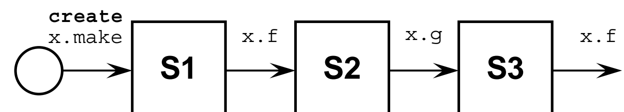

Figure 3. Lifecycle of an object

3. DISTRIBUTED COMPUTATION WITH SCOOP

When writing SCOOP-based applications, programmers can rely on the high-level concepts of *separate object* and *processor*, without taking into account the mapping of processors to the actual physical resources. This is one of the strongest ideas brought by SCOOP: it allows developers to write applications which would run both on a local machine (see Figure 4) and on several machines distributed over a local network or Internet (see Figure 5), without the need to make any changes in the software text. The mapping of processors to physical resources is not specified by the software text, hence the facility of executing SCOOP-based applications in a distributed setting.

In our example, `Processors 1`, `2`, and `3`, which are located on the same machine `Computer 1` may be also easily placed on three different machines: `Processor 1` may be handled by the computer `susi.ethz.ch`, `Processor 2` by the computer `ruth.ethz.ch`, and `Processor 3` may remain on the local machine `Computer 1`. The objects handled by each processor become thereby physically distributed. Nevertheless, from the application's point of view, as well as from the programmer's point of view, there is no difference between the two situations depicted by Figures 4 and 5. The SCOOP mechanism takes care of all the "dirty work", that is to say for making the distributed architecture completely transparent for the application, so that it can run as if all processors were located on the same machine.

[8] A multiple locking policy for SCOOP has been proposed in [11].

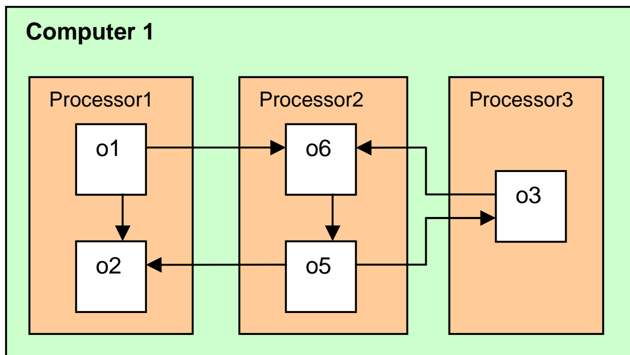

Figure 4. SCOOP application on a single machine

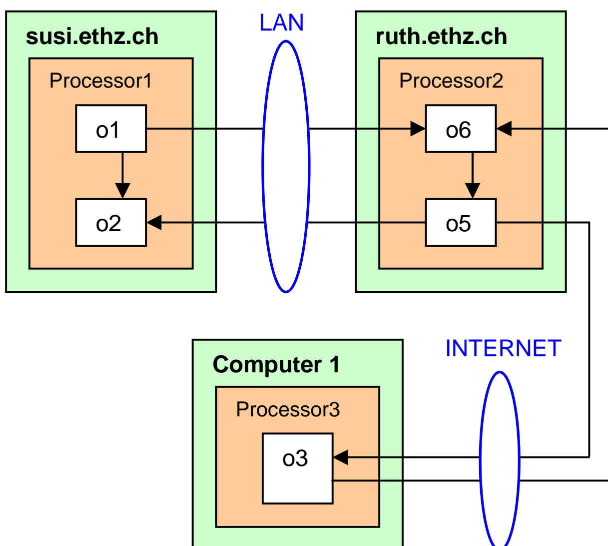

Figure 5. SCOOP application on distributed machines

3.1 Concurrency Control File

The Concurrency Control File (CCF) specifies the mapping of processors to actual physical resources: machines, processes, threads, application domains, web services, etc. The CCF file is separated from the software text. What's more, it is not a compulsory part of a SCOOP-based application. If the CCF exists, the mapping of the processors is done according to the information in the file. Should CCF be not available, a default mapping scheme is used[9]. The compilation of a concurrent application is completely independent from the existence or non-existence of a CCF. This means that the mapping to physical resources is done at run time[10].

Example 5. Concurrency Control File

```
creation
  system
    "susi.ethz.ch" (2): "c:/prog/appl1.exe"
    "ruth.ethz.ch" (4): "c:/prog/appl2.dll"
    Current: "c:/prog/appl1.exe"
  end
end
external
  db_handler: "schlemmer.ethz.ch" port 9000
  web_handler: "papi.ethz.ch" port 8080
end
default
  port: 8001; instance: 10
end
```

The `creation` part specifies which physical resources should be used for separate creations of the form `create x.make`, where `x` is separate. In the example above:

- two processors will be created at location represented by the application `appl1.exe` on the computer `susi.ethz.ch`,
- the next four processors will be created at location `appl2.dll` on the computer `ruth.ethz.ch`,
- the following ten will be created on the local computer. Value `10` comes from the `instance` entry in the `default` part of the CCF.

The allocation scheme is repeated for further separate object creations, starting again with two separate objects on the computer `susi.ethz.ch`, four on `ruth.ethz.ch`, and so on.

Sometimes, applications need references to separate objects previously created by another program. In such cases, the application relies on the CCF: the `external` part of the CCF specifies which external services should be used for requesting a reference to (persistent) separate objects. These may include database handlers, web services, ftp servers, etc. In our example, the location of database handler `schlemmer.ethz.ch` and web handler `papi.ethz.ch` is specified. The application can get a reference to a separate database object by using an appropriate function from facility class `CONCURRENCY` (see section 4):

`server (name:STRING; ...): separate DATABASE` with the argument "db_handler".

Finally, the `default` part of the CCF provides default values for SCOOP-related parameters used by the application, e.g. the standard port number for creation of a new processor, the default number of processors to be created, etc.

4. LIBRARY MECHANISMS

We have strived for the simplicity and elegance of the proposed model. The way to achieve these goals has been to keep the basic mechanism as minimal as possible. Naturally, users' needs cannot

---

[9] For example, on the .NET platform each new processor is, by default, mapped to a new *application domain* on the local machine [12].

[10] See section 4.1 for more details.

be fully predicted, that is why SCOOP offers several library mechanisms that extend the capabilities of the mechanism. In this section, we describe two of them:

- the physical resource management, which allows for a finer degree of resource control than the CCF files,
- the mechanism of duels, which may serve as a basis for real-time programming.

Both mechanisms may be implemented as a facility class CONCURRENCY [8].

4.1 Resource management

With a CCF-like approach (see 3.1), the application software will, most of the time, not concern itself with the mapping of processors to the physical concurrency architecture. Some application developers may, however, need to exert a finer degree of control from within the application, at the possible expense of dynamic reconfigurability. Some CCF functionalities must then be accessible directly to the application, enabling it, for example, to select a specific process or thread for a certain processor. They will be available through libraries as part of the two-level concurrency architecture [8].

At the other extreme, some applications may want unlimited run-time reconfigurability. It is not enough to read a CCF at start-up time and then be stuck with it. But we cannot either expect to re-read the configuration file before each operation, as this would kill performance. The solution is once again to use a library mechanism: a procedure must be available to read or re-read the configuration information dynamically, allowing the application to adapt to a new configuration when (and only when) it is ready to do so.

4.2 Duels

SCOOP allows the processor in charge of an object to execute at most one routine at any given time (see section 2.5). The main reason for this restriction is to retain the ability to reason about our software. Nevertheless, in cases of emergency, or if a client keeps a lock on a supplier object for too long, it should be possible to interrupt the client by triggering an exception. This observation has inspired us to introduce the mechanism of *duels*. A duel can be defined as an attempt to snatch a shared object from its current holder. Let us illustrate this with an example. An object executes the instruction `r(b)`, where `b` is separate. After having waited for the object `b` to become free, and for the separate precondition clauses to hold, the object captures `b`, and becomes its current *holder*. It should be emphasised that the execution of `r` on `b` has started on behalf of the holder, but is not finished. Another separate object, which we call the *challenger,* executes `s(c)`, where `c` is separate and attached to the same object as `b`. In normal case, the challenger has to wait until the call to `r` is over. If the challenger is in hurry or has something more important to do with the reserved object, it can use the facilities of the library class `CONCURRENCY` to snatch the reserved object. Class `CONCURRENCY` provides, for this purpose, features `yield` and `retain` for the holder and features `demand` and `insist` for the challenger (see Table 1). On the holder's side, `yield` means that the holder is ready to release his hold if a challenger with a more urgent need comes along. By calling the feature `retain`, which is the default behaviour, the holder can retain his hold. On the challenger's side, `demand` means that the challenger wants to get immediately the hold. If the holder has called `yield` then the challenger will indeed get the hold, and the holder will get an exception, otherwise the challenger will get an exception. On the contrary, `insist` on the challenger's side means that the challenger tries to get the hold, but if the holder has called `retain`, the challenger will not get an exception; it will simply wait. Otherwise, that is if the holder has called `yield`, the challenger will get the hold, and the holder will get an exception. For the challenger to return to the default behaviour of waiting for the holder to finish, `wait_turn` is used. A call to one of these features will retain its effect until another one supersedes it. It should be emphasised that the two sets of facilities are not exclusive; for example a challenger could use both `insist` to request special treatment and `yield` to accept being interrupted by another client. Table 1 summarises the results of a duel in all possible cases. Default options and behaviour are underlined.

Table 1. Sematics of duels

Challenger → ↓ Holder	**`wait_turn`**	**`demand`**	**`insist`**
`retain`	Challenger waits	Exception in challenger	Challenger waits
`yield`	Challenger waits	Exception in holder's routine	Exception in holder's routine; serve challenger

We plan to extend the duel mechanism with a priority scheme, in order to support advanced real-time programming.

5. DISCUSSION

5.1 Design criteria

In section 1, we mentioned several criteria that should guide the design and implementation of an object-oriented concurrency mechanism. Let us discuss these criteria in more detail and see how the SCOOP model fulfills them.

5.1.1 Minimality of mechanism

Object-oriented software construction is a rich and powerful paradigm, which intuitively seems ready to support concurrency. It is essential, then, to aim for the smallest possible extension. Minimalism here is not just a question of good language design. If the concurrent extension is not minimal, some concurrency constructs will be redundant with the object-oriented constructs, or will conflict with them, making the developer's task hard or impossible. To avoid such a situation, we must find the smallest syntactic and semantic *epsilon* that will add concurrent execution capabilities to our object-oriented programs.

The extension presented in this article is indeed minimal, since it is not possible to add less than one new keyword.

5.1.2 Full use of object-oriented techniques

It would be unacceptable to have a concurrent object-oriented mechanism that does not take advantage of all object-oriented techniques, in particular inheritance. Please note that the "inheritance anomaly" [7], which causes so much trouble in the approaches proposed by other authors, as well as other potential

conflicts, are not inherent to concurrent object-oriented development but follow from specific choices of concurrency mechanisms, in particular active objects, state-based models and path-expressions-like synchronisation; the appropriate conclusion is to reject these choices and retain inheritance.

The SCOOP model allows the programmer to use the full power of all the object-oriented techniques offered by the underlying language Eiffel, including multiple inheritance, genericity (see 5.1.5), agent mechanism, and information hiding.

5.1.3 Compatibility with Design by Contract

It is essential to retain the systematic, logic-based approach to software construction and documentation expressed by the principles of Design by Contract.

A fundamental place in the SCOOP mechanism has been given to assertions, in particular preconditions. In fact, the model is largely derived from the analysis of the new semantics of preconditions in a concurrent setting (see 2.4).

5.1.4 Applicability to many forms of concurrency

A general criterion for the design of a concurrent mechanism is that it should support many different forms of concurrency: shared memory, multitasking, client-server computing, distributed processing, real-time programming, etc.

With such a broad set of application areas, a language mechanism cannot be expected to provide all the answers. Nevertheless, it should lend itself to adaptation to all the intended forms of concurrency. In SCOOP, this is achieved by using the abstract notion of processor, and relying on a distinct facility (Concurrency Control File, libraries) to adapt the solution to any particular hardware architecture that may be available.

5.1.5 Support for the reuse of non-concurrent software

It is necessary to support the reuse of existing, non-concurrent software, especially libraries of reusable software components.

SCOOP allows for a smooth transition between sequential classes such as `QUEUE [G]` [11] and their concurrent counterparts such as `BUFFER [G]`. The latter, which we have used in several code examples within this article (see Example 1, 2, and 3), can be simply defined as:

```
separate class BUFFER [G]
  inherit QUEUE [G]
end
```

Sometimes, as an inescapable consequence of the semantic differences between sequential and concurrent computation, some wrapper classes may be needed but, in most cases, they are very easy to write.

5.1.6 Efficient deadlock-avoidance scheme

One area in which more work remains necessary is how to guarantee deadlock avoidance. Deadlock potential is a fact in a concurrent life. For example any mechanism that can be used to program semaphores can cause deadlock, since semaphores are trivially open to that possibility.

The solution lies partly in the use of high-level encapsulation mechanisms. For example a set of classes encapsulating semaphores should come with behaviour classes that automatically provide a *free* operation for every *reserve*, thereby guaranteeing deadlock avoidance for applications that follow the recommended practice by inheriting from the behaviour class.

This approach may be insufficient, however, and it is advisable to design simple anti-dealock rules, automatically checkable by static tools (at compilation time). Devising an efficient deadlock-avoidance scheme is one of our principal goals.

5.1.7 Adaptability through libraries

Many concurrency mechanisms have been proposed over the years (see section 5.3). Each has its partisans, and each may provide the best approach to certain problem areas. It is important, then, that the proposed model should support at least some of these mechanisms. More precisely, the solution must be general enough to allow us to program various concurrency constructs.

One of the most important aspects of the SCOOP model is that it supports the construction of libraries for widely used schemes. The library construction facilities (classes, assertions, genericity, multiple inheritance, deferred classes and others) allow us to express many concurrency mechanisms in the form of library components. We expect that a number of libraries will be produced, relying on the basic tools and complementing them, to support concurrency models catering to specific needs and tastes.

In section 4, we have also seen the use of library classes such as `CONCURRENCY` to provide various refinements to the basic scheme defined by the language mechanism.

5.2 Implementation

SCOOP has only had prototype implementations so far. One of our principal goals is to provide working, production-quality implementations of SCOOP on several platforms.

The first implementation [12] takes advantage of the Remoting library of the new .NET framework, available today from Microsoft and in the process of being ported to non-Microsoft architectures, in particular Linux and BSD in two separate open-source implementations. .NET is attractive as an infrastructure for several reasons: the general quality of its design; its support for multi-language interoperability, so that components developed in one language can be made available to many others; its innovative solutions in the area of Internet programming, especially its support for Web services; and particularly the power of the Remoting library that provides a general basis which appears to match particularly well the needs of the SCOOP model.

Other targeted platform include the POSIX threading library and the .NET Compact Framework.

5.3 Related work by other authors

Hewitt and Agha's *Actors* model [1], which predates the object-oriented renaissance and comes from a somewhat different background, has influenced many object-oriented approaches. Actors are computational agents similar to active objects, each with a mail address and a behaviour. An actor communicates with others through messages sent to their mail addresses; to achieve

[11] `QUEUE [G]` is a *generic class*. `G` is the *formal generic parameter*, which should be replaced by an *actual generic parameter*, e.g. `INTEGER`.

asynchronous communication, the messages are buffered. An actor processes messages through functions and by providing "replacement behaviours" to be used in lieu of the actor's earlier behaviour after a certain message has been processed.

One of the earliest and most thoroughly explored parallel object-oriented languages is POOL [2]; POOL uses a notion of active object, which was found to raise problems when combined with inheritance. For that reason inheritance was introduced into the language only after a detailed study which led to the separation of inheritance and subtyping mechanisms. The design of POOL is also notable for having shown, from the start, a strong concern for formal language specification.

[21] contains the description of several influential Japanese developments, such as ABCL/1. MUSE, an object-oriented operating system developed at Sony Computer Science Laboratory, was presented by Tokoro and his colleagues at TOOLS Europe 1989 [20]. The term "inheritance anomaly" was introduced by Matsuoka and Yonezawa [7], and further papers by Matsuoka and collaborators which propose various remedies.

Work on distributed systems has been particularly active in France, with the GUIDE language and system [3] and the SOS system [17]. In the area of programming massively parallel architectures, primarily for scientific applications, the EPEE system has been developed [6].

Also influential has been the work done by Nierstrasz and his colleagues at the University of Geneva around the Hybrid language [13][15], which does not have use two categories of objects (active and passive) but relies instead on the notion of thread of control, called *activity*. The basic communication mechanism is the remote procedure call (RPC), either synchronous or asynchronous.

A special issue of the *Communications of the ACM* [9] presents a number of important approaches to concurrent object-oriented programming, originally drawn from concurrency papers at various TOOLS conferences. An earlier collective book edited by Yonezawa and Tokoro [21] served as catalyst for much of the work in the field and is still good reading.

There are several survey articles on aspects of concurrent object-oriented languages. [14] discusses systems providing process or object migration in a distributed context. [19] studies several languages and discuss whether the concurrency is appropriately integrated into them. [15] gives a first classification of concurrent object-oriented languages. Large surveys on COOLs are provided by [18] and [16].

6. CONCLUSIONS AND FUTURE WORK

We have presented the basic concepts of the SCOOP model. We introduced the notion of processor, and proposed an object-oriented concurrency mechanism based on that concept. We have discussed the semantics of contracts in a non-sequential setting and their use for devising an access control policy for SCOOP. We have shown how high-level concurrency constructs are mapped to the physical concurrency architecture, and discussed distributed programming with SCOOP. We have also presented the library mechanisms of the model.

The main contribution of our work is the cohesive description of SCOOP: a compact yet powerful model for object-oriented concurrent programming. Unlike other models, SCOOP is based on very high-level concepts (processors and separate objects), which make it possible to keep the full power of all object-oriented techniques, and to apply the model to many different forms of parallel programming. SCOOP makes concurrent and distributed programming much easier: programmers can forget the usual "concurrency nightmares" such as semaphores, monitors, and locks. It also allows us to produce software that runs on very different physical configurations (single machine, several machines on a local network, machines distributed over Internet) with no need to change the code (or re-compile it) each time the configuration changes.

Another important result is an in-depth analysis of the semantics of contracts in a concurrent setting. This has allowed us to confirm the importance of Design by Contract as a key technique for obtaining high-quality software.

We are currently implementing the SCOOP model on .NET [12]. We are also devising a new access control policy based on the concept of pure function [11]. We are interested in providing a mathematical model of execution of SCOOP-based programs, which would permit to prove the correctness of SCOOP-based programs and to define an efficient deadlock-prevention policy. Finally, one of the main research topics is the application of SCOOP to real-time systems. This involves an extension of the duel mechanism with a priority scheme, and devising timing assertions.

7. ACKNOWLEDGMENTS

The research work presented in this paper is part of the project "SCOOP: Environment for dependable distributed and reliable object-oriented computing, based on the principles of Design by Contract". This project has been financially supported by the Hasler Foundation (Berne, Switzerland).

8. REFERENCES

[1] Agha, G. Concurrent Object-Oriented Programming, in Communications of the ACM, 1990, 33(9), 125-141

[2] America, P., Beemster, M. A portable implementation of the language POOL, in Proceedings of TOOLS EUROPE 1989, ed. Jean Bézivin, SOL, Paris, 1989, 347-353

[3] Balter, R. et al. Architecture and Implementation of Guide, an Object-Oriented Distributed System, in Computing Systems, 1991, vol. 4

[4] Caromel, D. Towards a Method of Object-Oriented Concurrent Programming, in Communications of the ACM, Volume 36, Number 9, September 1993, 90-102

[5] Garcia-Molina, H., Ullman, J. D., Widom, J.D. Database Systems: The Complete Book, Prentice Hall, 2002

[6] Jézéquel, J.-M. Object-Oriented Software Engineering with Eiffel, Addison-Wesley, Reading (Mass.), 1996, chapter 9

[7] Matsuoka S., Yonezawa A. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in Agha G., Wenger P., Yonezawa A. (eds.), Research Directions in Concurrent Object-Oriented Programming, MIT Press, Cambridge (Mass.), 1993, 107-150

[8] Meyer, B. Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997

[9] Meyer B. (ed.), Special issue on Concurrent Object-Oriented Programming, in Communications of the ACM, 1993, 36(9)

[10] Meyer B. Systematic Concurrent Object-Oriented Programming, in Communications of the ACM, Special issue on Concurrent Object-Oriented Programming, 1993, 36(9), 56-80

[11] Nienaltowski, P. Extending the access control policy for SCOOP, submitted for publication, available online at http://se.inf.ethz.ch/people/nienaltowski/extended_access_control_draft.pdf

[12] Nienaltowski, P., Arslan, V. SCOOPLI: a library for concurrent object-oriented programming on .NET, in Proceedings of the 1st International Workshop on C# and .NET Technologies for Algorithms, Computer Graphics, Visualization, Scientific, Distributed and Web Computing, 2003

[13] Nierstrasz, O. A Tour of Hybrid: A Language for programming with Active Objects, in Advances on Object-Oriented Software Engineering, Meyer B., Mandrioli D. (eds.), Prentice-Hall, 1992, 167-182

[14] Nuttal, M. A brief survey of systems providing process or object migration facilities, Operating Systems Review, 1994, 28(4), 64-80

[15] Papathomas, M. Language design rationale and semantic framework for concurrent object-oriented programming, PhD Thesis, University of Geneva, Switzerland, 1992

[16] Philippsen, M. A survey of concurrent object-oriented languages, Concurrency: Practice and Experience, 2000, 12, 917-980

[17] Shapiro, M., Gautron, P., Mosseri, L. Persistence and Migration for C++ Objects, in ECOOP 1989, ed. Cook, S., Cambridge University Press, Cambridge (England), 191-204

[18] Turcotte, L.H. *A survey of software environments for exploiting network computing resources,* Technical Report, Mississippi State University, 1993

[19] Wyatt, B., Kavi, K., Hufnagel, S. Parallelism in object-oriented languages: a survey, IEEE Computer, 1992, 11(6), 56-66

[20] Yokote, Y., Teraoka, F., Yamada, M., Tezuka, H., Tokoro, M. The Design and Implementation of the MUSE Object-Oriented Distributed Operating System, in TOOLS 1, Bézivin J. (ed.), SOL, Paris, 1989, 363-370

[21] Yonezawa A., Tokoro M., eds. Object-Oriented Concurrent Programming, MIT Press, Cambridge (Massachussetts), 1987