

Statically Safe Intra-Object Concurrency

Miguel Oliveira e Silva
Departamento de Electrónica e Telecomunicações – IEETA
Universidade de Aveiro
Aveiro, Portugal
mos@det.ua.pt

ABSTRACT

This article analyses and proposes the static safe integration in static typed concurrent object-oriented languages of several synchronization schemes, ranging from the conservative monitor to the most liberal lock-free scheme. Mixed synchronization schemes are also presented. The problem of the realizability by programming languages compiling systems of all those schemes, is also approached in some detail. Care is taken to ensure static safety of all the presented approaches.

To simplify an objective comparison between different synchronization schemes, a concurrent object availability metric is also proposed and used.

The problem of synchronism scheme selection is also briefly approached.

This work is part of the development of a concurrent object-oriented language, MP-EIFFEL, strongly based on the language EIFFEL.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features—*Abstract data types, Classes and objects, Concurrent programming structures*; D.3.2 [Programming Languages]: Language Classifications—*Concurrent, distributed, and parallel languages, Object-oriented languages*

General Terms

Languages, Theory, Algorithms

Keywords

Intra-Object Concurrency, Static Safety, Synchronization schemes, Nonblocking synchronization, Lock-free, COA – concurrent object availability, Eiffel

1. INTRODUCTION

Integrating concurrent programming in object-oriented languages has proved to be a difficult task. Not only concurrency may interfere with other language constructs (for example, inheritance [26]), but it may also compromise, in the absence of correct object synchronization, the simple and essential interpretation of objects as abstract data types.

On the other hand, a vast group of synchronization schemes are available that can be applied to protect and allow the access to shared resources. Usually, concurrent object-oriented languages stick to a predefined conservative synchronization scheme (such as monitors). Such schemes simplify the assurance of intra-object safety, but they reduce the concurrency potential of the program (resulting from an increased processor blocking and contention), limit its adaptability to different application areas, and also, most importantly, may compromise some systemic safety issues, such as the inexistence of deadlocks. Hence, enhancing the ability of several processors to operate concurrently inside objects is a desired goal, as long as the safety of the programming language is not compromised.

This article analyses and proposes static approaches for the integration of different safe intra-object synchronization schemes in concurrent languages. The identification of the requirements posed by each scheme and its realizability by a compiling system are also relevant results of this article.

Other safety problems resulting from the integration of concurrency in object-oriented languages, such as interferences with the inheritance mechanism [26], are beyond the scope of this article. Some of them have been approached elsewhere [6].

This work, is part of the development of a static typed concurrent object-oriented language, named MP-EIFFEL¹ [6].

2. BASIC DEFINITIONS

For a better understanding of this article, some basic terms and definitions are presented here.

2.1 Abstract concurrent programming

This article is about general concurrent programming, hence we don't want to attach the concurrent processing entities to specific realizations, such as *threads*, processes, or any

¹Multi-Processor Eiffel.

other. Instead we will use an abstract notion of “processor” adapted from Meyer [29, page 964]:

A processor is an autonomous thread of control capable of supporting the sequential execution of instructions.

2.2 Safety

On language design Hoare [18] presents safety² as an essential rule for language constructs:

Programming language mechanisms should be prevented from producing meaningless results.

Concurrent programming systems raise specific safety problems, such as race conditions, deadlocks and processor starvation.

Race conditions may arise when several processors attempt to concurrently modify a shared resource without a proper synchronization scheme.

Deadlocks are systemic synchronization problems resulting from situations in which processors wait forever for each others allocated resources. One of the (four) conditions responsible for its appearance is processor locking [4]. From synchronization schemes requiring less processor blocking, results lower likelihood of deadlocks, or even its complete prevention. Deadlocks are part of the more general problem of ensuring liveness in concurrent programs [24].

A safe concurrent programming language should not allow the existence of any of these safety problems. Liveness problems are sometimes difficult to prevent, and so hard to solve. However, that is not the case of race conditions. They have simple solutions, requiring only the correct use of a synchronization scheme. A statically safe concurrent language must ensure, at compile time, their inexistence at run time.

2.3 Concurrent Objects

The term “concurrent object” will be used throughout this article to mean objects that may be used by more than one processor. In MP-EIFFEL there are two types of concurrent objects with different, but well defined semantics: **remote** and **shared**, hence the choice for this terminology instead of the more commonly used: shared object.

2.4 Concurrent Object Availability

In order to compare different synchronization schemes, it is useful to have some kind of objective metrics expressing the ability for an object to be executed concurrently. That is the purpose of the Concurrent Object Availability metric.

Considering that N_x is the maximum number of processors with some property x (for example, the reading or writing properties) which may use a concurrent object OBJ , and that N_c is the maximum number of such processors which can concurrently operate inside OBJ (of course: $N_c \leq$

²Hoare uses the term “security” for this purpose.

N_x), we define the Concurrent Object Availability of OBJ in relation to the processors with the x property as being:

$$COA_x = \frac{N_c}{N_x} \quad (1)$$

This factor measures the maximum percentage of processors with some property that can safely operate concurrently inside an object.

It should be mentioned that this factor may not be unique in each synchronization scheme, and may depend on the concurrent state of the object (for example, the use of an object by processors with a certain property may exclude its usage by other type of processors).

3. CORRECTNESS CONDITIONS

Object-Oriented software construction is defined by Meyer [29, page 147] as the building of software systems as structured collections of possibly partial abstract data type (ADT) implementations. Therefore, the correctness of an object-oriented program depends mainly on the correctness of each of the ADT’s it implements, regardless of the possible complex interactions they might have between them.

Liskov and Zilles [25] define an ADT as being a class of objects which is completely defined by the external operations available on them. This definition was extended to take into consideration the essential semantic properties of each ADT [7, 20, 29].

Viewing an object as an instance of a ADT implementation simplifies how it is used, and provides a solid theoretical basis for object-oriented programming.

In sequential programs, in order not to compromise the correctness and simplicity of the ADT, it is necessary to allow the use of objects only in their stable times [29, page 364], and to forbid the existence of public modifiable attributes (otherwise, any client could change the object’s state outside of its control).

However, in concurrent programs under the presence of intra-object concurrency (more than one processor active within an object) the problem is much more complex. In these cases in order to ensure the correctness of an object it is necessary to observe the following rule:

Object Concurrent Integrity

Intra-object concurrency cannot, in any case, compromise the implementation of the ADT of its class.

This rule ensures that, from the point of view of each object, its integrity and correctness is not compromised if used concurrently. However, this rule is not sufficient to ensure that the correctness of the programs themselves is not compromised. It is also necessary to ensure that the logical order of actions imposed by the sequential program of each processor, is not changed. It wouldn’t be acceptable if a rearrangement of the actions of one processor would result in

a sequence of actions not equivalent to the logical sequence causality imposed by the processor's sequential program.

Processor Concurrent Integrity

Intra-object concurrency cannot, in any way, compromise the logical causality of actions imposed by the sequential programs of each processor.

3.1 Sequential Consistency

Both rules are very similar to the sequential consistency condition defined by Lamport in the context of multiprocessor systems [23].

Sequential Consistency

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

If we apply this condition to objects, and to their services – considering that they are attached to the applicable ADT semantic conditions (class invariants, and service preconditions and postconditions) – as the operations to which the sequential consistency is to be observed, it is obvious that this condition is sufficient (and necessary) to verify the two correctness rules presented in the beginning of this section.

The applicability of this condition to concurrent objects has been stated long ago by other authors [17].

3.2 Total Object Covering

An immediate necessary restriction resulting from the sequential consistency condition is the requirement that the object's synchronization scheme covers all of its external services. In the absence of this restriction, most likely race conditions would arise on the access of the object's attributes, compromising, in an unpredictable way, the correctness of the object's ADT implementation.

Total Object Covering

A correctness condition for the synchronization of concurrent objects is the necessity that all of the object's exported services are protected with a synchronization mechanism.

One of the strongest objections [10] to the concurrent mechanisms of the language JAVA is the fact that its programs may not observe this rule (failing to properly implement monitors).

3.3 Linearizability

An important particular case of sequential consistency, is the linearizability condition [16, 17].

Linearizability

An object is linearizable if each operation appears to take effect instantaneously at some point between the operation invocation and response.

Whereas the sequential consistency condition only restricts the causality of operations in each processor (allowing an arbitrary reordering in the processing of operations of different processors), the linearizability condition imposes also the restriction of causality (in the above terms) between operations of different processors. If an operation O_1 executed by a processor, finishes before an operation O_2 to be executed by another processor begins, then this causality has to be maintained.

Unlike the sequential consistency condition, linearizability has the property of being local [17]. So, the verification of the linearizability of each object is enough to ensure that a program is also linearizable.

Another important property of this correctness condition is the fact that it does not impose object blocking. This criterion creates the possibility of using in a safe way lock-free synchronization schemes, reducing, or even eliminating, the risk of program deadlocks or processor starvation.

4. SYNCHRONIZATION SCHEMES

Having defined the essential correctness conditions to observe in the synchronization of concurrent objects, in this section we will present several synchronization schemes, and the restrictions they pose on the programming language compiling system in order to allow a statically safe implementation.

4.1 Synchronizing through ADT assertions

In languages that allow the explicit declaration of runnable assertions attached to classes (invariants) and their services (preconditions and postconditions) to axiomatically express the ADT semantics, such as EIFFEL, we might be tempted – when taking into consideration the basic correctness condition of not compromising the ADT when synchronizing objects – to use these assertions, specially invariants, as synchronization guards to control intra-object concurrency. However, this approach – although theoretically appealing – is not safe since it does not ensure the inexistence of race conditions. It wouldn't be acceptable that an object has a different concurrent behavior depending only on a more complete, or incomplete, definition of the invariant of its class. It is frequent, in EIFFEL, to express a part of the invariant as a comment, or not to express it at all. This omission, obviously, does not mean that we are dealing with different invariants, only that we were lazy or unable to completely express it. For example, in the implementation of the ADT of a STACK in EIFFEL, usually the essential *Last In First Out* property is not expressed in the invariant, although one expects its observance.

Since we are looking for statically safe synchronization schemes this approach is therefore, not acceptable.

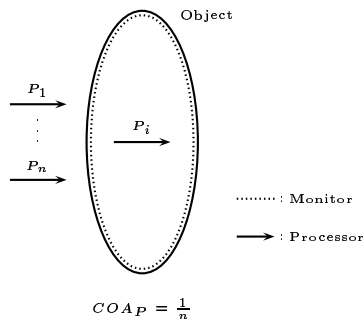


Figure 1: Monitors

4.2 Monitors

A much simpler and sufficient approach to ensure sequential consistency, is to consider each object to be a monitor (figure 1). Hoare [19] and Hansen [9] themselves have recognized the importance of the class concept of the first object-oriented language, SIMULA, when they proposed the monitor synchronization scheme.

Monitors ensure mutual exclusion of processors on the use of the object, hence the sequentiality of operations within objects is preserved. However, objects synchronized with a monitor are only available to one processor at a time. For n processors the monitor COA value is $\frac{1}{n}$, which is its lowest possible useful value.

Several programming languages use monitors as the essential synchronization scheme, such as the family of languages based on the ACTORS model [1], and the concurrent extension to EIFFEL named SCOOP³ [28, 29].

The language JAVA failed to properly implement monitors, since it leaves to the programmer the responsibility of correctly synchronizing the object's services (to which it provides the **synchronized** keyword). Hence it is not a statically safe language for concurrent programming [10].

4.2.1 Realizability

A safe implementation of monitors by the compiling system raises few problems. When compiling a program, the compiling system only needs to know which objects are shared by several processors (not necessary the classes, since it is possible for a class to have sequential and concurrent instances, as happens with MP-EIFFEL).

A possible language approach sufficient for a safe identification of concurrent objects, is to use the language type system, extending it with appropriate concurrency annotations. This is the approach followed by several languages such as, for example: CONCURRENT PASCAL⁴ [8], SCOOP, and MP-EIFFEL. The implementation of this behavior in the code generation is straightforward: all that is required is to implement concurrent objects as monitors⁵.

A safety requirement imposed by monitors – and also by

³Simple Concurrent Object-Oriented Programming.

⁴Although this language is not object-oriented

⁵The implementation of conditional synchronization in the monitor is dealt elsewhere [6]

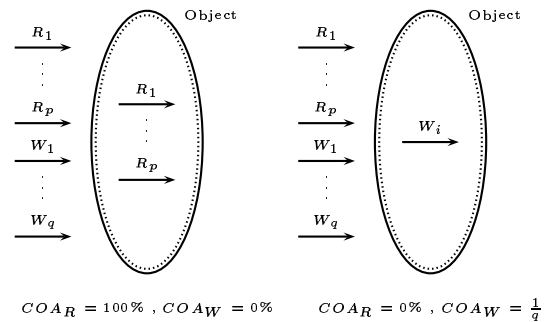


Figure 2: Readers-Writer Exclusion

other synchronization schemes as it will be seen – is the imposition of not allowing a direct external modification of any of the object's attributes (this imposition is also necessary in order to allow an appropriate ADT implementation). If this condition is not verified, then the synchronization implementation of each object ceases to be encapsulated inside the object boundaries, and has to be propagated to all the places in the program in which the object is used in this unacceptable way. Furthermore, there are possible interferences with the inheritance mechanism when this condition is not verified. This requirement is a basic condition in order to avoid the propagation beyond the object's boundaries, of a safe implementation of monitors.

The languages C++ and JAVA do not verify this condition, since they allow a public declaration of attributes in classes. EIFFEL also allows the public declaration of attributes, but external clients cannot modify the attributes, so the problem does not occur.

4.3 Readers-Writer Exclusion

The imposition of mutual exclusion for processors requiring the execution of services in concurrent objects, may be considered an overwhelming restriction. Frequently, some of the processors are only trying to query (without side-effects) the object to get some information. In these cases, it is sufficient to ensure mutual exclusion only when a service with side-effects on the object state (usually a procedure encapsulating a command request) is being processed, allowing the concurrent processing of the remaining (query) services.

Hence an approach using the synchronization scheme of readers-writer exclusion [5] (a “writer” processor⁶ excludes all the others, but multiple “reader” processor⁷ can operate concurrently) is also a valid and safe choice (figure 2). This scheme has higher average COA values than monitors, hence is less prone to block the access of concurrent objects, which may also reduce the risk of some global (program wide) liveliness problems such as deadlocks.

This synchronization scheme is used in the language ADA (in its recent protected types extension), and was also the first approach taken by the prototype language MP-EIFFEL, proposed by the author.

4.3.1 Realizability

⁶Executing an object service with possible side-effects.

⁷Executing an object service without side-effects.

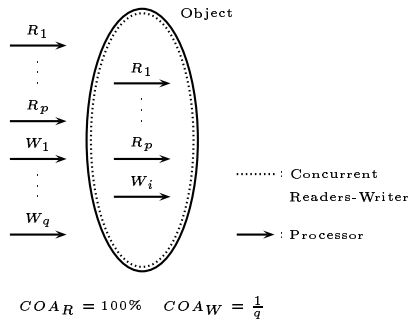


Figure 3: Concurrent Readers-Writer

This synchronization scheme is “better” (higher average COA values) than monitors, but the compiling system requires more information about the program. Besides the identification of concurrent objects, the compiling system will have to know how to separate services from the object’s classes⁸ which have side-effects (“writers”) from those which do not (“readers”).

The approach followed in MP-EIFFEL to identify the existence of side-effects in a service uses the following reasoning (possible because attributes can only be changed by the object’s services). A service has side-effects if its program includes an assignment instruction to one of the object’s attributes, or if there is a call to a routine with side-effects. Recursive routines (either directly or through other routines), pose no problem to this approach as long as the compiling system keeps track of the routines already traversed.

Again, for the sake of clarity, the type system can be extended with an annotation indicating that a service must be implemented without side-effects. However it is not necessary to have such a type annotation to safely implement this synchronization scheme.

4.4 Concurrent Readers-Writer

Lamport [22] has proposed a generalization to the last synchronization scheme which allows the concurrent access of multiple “readers” processors and a single “writer” processor. Mutual exclusion is only required to multiple “writer” processors (figure 3). In this way, “reader” processors never block a possible “writer” processor.

In Lamport’s proposal, in order to ensure that a “reader” processor is done in object stable times (when the invariant holds), the requested service (query) is repeated whenever it occurs concurrently with a “writer” operation.

In the integration of this scheme within objects it is necessary to foresee the situation of an invariant failure in the beginning of the execution of one, or more, “reader” services resulting simply due to a concurrent execution of a “writer” service. This possibility needs to be properly taken care, imposing, for example, the repetition of the “reader” services when the invariant fails and a concurrent “writer” access has been detected (otherwise, the invariant should indeed fail).

⁸Since all the instances of a class share the implementation of each service

This scheme is very interesting due to the fact that, in its implementation, it does not impose much more restrictions than those imposed by the previous scheme⁹ (in the Lamport approach, it only requires the possibility of repeating “reader” services). It has less contention (higher or equal COA value) in the execution of “writer” services which reduces the risk of deadlocks. However, it may create starvation problems in the “reader” services when the execution of “writer” services is overwhelmingly frequent [22, 30].

A possible solution, in some cases, to this problem is proposed by Peterson [30]. The main idea is based in the duplication of the object’s state.

In the important particular case in which there is only a unique processor with the possibility to execute “writer” services (situation which occurs in MP-EIFFEL), Peterson [30] proposes a wait free algorithm to any processor which makes the object always available ($COA = 100\%$) to any processor. Another more recent possibility is to use RCU¹⁰ algorithms [27].

4.4.1 Realizability

This synchronization scheme maintains the requirements imposed by the reader-write exclusion scheme, extending them with the necessity of allowing possible repetitions of “reader” services.

This repetition (hidden from the object’s clients) does not pose serious implementation and semantic problems because – by definition – “reader” services don’t change the object’s state. However, since the object state might change during “readers” execution due to a concurrent “writer” execution, there is the possibility of exceptions being raised, resulting from invariant, or other assertions, failure. Hence, this synchronization scheme requires a language in which it is possible to transparently catch all the exceptions created during the execution of services, allowing to verify if the failure cause was due to the interference of a concurrent “writer” service – in which case it can be ignored and the execution of the “reader” service has to be repeated – or if it is a real correctness failure. This restriction is essential to the implementation of this scheme, because it is the only way to distinguish real failures from harmless race-condition ones.

This problem can be completely avoided in the particular case where there is only one processor with permission to invoke “writer” services. In this situation there are algorithms, such as the one proposed by Peterson [30], in which “reader” processors always observe the concurrent object at stable times, also with the extra advantage of those algorithms being wait-free.

MP-EIFFEL makes full use of this particularity since it allows a side-effect free usage (remote objects) of concurrent objects.

4.5 Lock-Free Synchronization

A group of synchronization schemes that has been deserving a growing enthusiastic interest are lock-free and wait-free

⁹Readers-Writer Exclusion

¹⁰Read-Copy Update

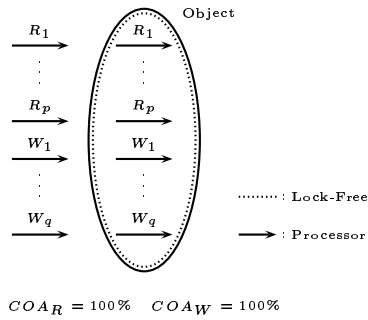


Figure 4: Lock-Free Synchronization

synchronization [13] (figure 4). This type of synchronism is characterized by the assurance that processors are able to execute operations on shared resources, regardless of the execution time of other processors, always with the guarantee that at least one of them will always be successful (an important particular case is wait-free synchronization, in which it is ensured that all processors will be able to perform the requested operation in finite time).

The advantages of this approach are the inexistence of processor blocking (making it immune to deadlocks) and tolerance to faults on other processors. These characteristics make it especially suitable for real-time programming [2].

Currently, this type of synchronism is seldom used, though some changes are expected in the future. For example, a library of classes¹¹ using this type of synchronism was recently released for JAVA,

The reasons why lock-free synchronization is so rarely used are its complexity, the specificity and low level of many of its algorithms, and also because it is difficult to ensure safe implementations. In our opinion the definition and construction of concurrent programming language mechanisms cannot ignore these synchronization schemes, and should be adaptable to their integration, or else they could become obsolete in the future. The challenge is, nevertheless, hard: how to integrate at compile time lock-free synchronization schemes within concurrent languages mechanisms without compromising safety?

4.5.1 Basic Concepts

In general, lock-free synchronization algorithms are based on the total, or partial, duplication of the concurrent object's attributes and, when necessary, in concentrating all of the necessary modifications to the object required by any of its services in a unique atomic instance. Usually this atomic object state modification is implemented using special hardware instructions, such as the instructions CAS (*Compare-And-Swap*) or LL/SC (*Load-Linked, Store-Conditional*). In those algorithms it is necessary to accept a possible failure (due to the action of another concurrent processor). When this happens, it is necessary to repeat the entire process. In the special case of wait-free algorithms, as mentioned before, a limit to the maximum number of repetitions is ensured.

Herlihy [12, 13] has demonstrated that there are universal al-

¹¹JSR 166: Concurrency Utilities.

```

1. fail = true;
2. do
3. {
4.   obj_cpy.copy(obj);
5.   if (obj_cpy.copy_succeed(obj))
6.   {
7.     obj_cpy.command(...);
8.     fail = !obj_atomic_replace_on_linearizability(obj_cpy);
9.   }
10. } while(fail);

```

Figure 5: Generic “writer” lock-free algorithm

```

1. fail = true;
2. do
3. {
4.   obj_cpy.copy(obj);
5.   if (obj_cpy.copy_succeed(obj))
6.   {
7.     result = obj_cpy.query(...);
8.     fail = false;
9.   }
10. } while(fail);

```

Figure 6: Generic “reader” lock-free algorithm

gorithms able to implement this synchronism in concurrent objects¹² observing the linearizability condition, presenting also universal methodologies (though not very efficient) [12, 14] for its implementation. The presented methodology, as mentioned by Herlihy, is adaptable to be automatically executed by the compiling system.

It should be mentioned that the majority of lock and wait-free algorithms are adapted to specific data structures (such as *stacks*, *queues* and linked lists) and make extended use of the specific and detailed knowledge about the behavior of those data structures. An automatic implementation using such profound knowledge of the ADT of objects, currently does not seem viable, since it would require an advanced comprehension ability by the compiling system in order to take advantage of it. So we are left with the existent possibilities for syntactic approaches to the problem, that ensure the necessary correctness conditions.

There are some attempts to define lock-free language constructs such as the proposal of Harris and Fraser [11] to extend JAVA with a new atomic instruction.

4.5.2 Realizability

Generic algorithms [14] for this synchronization scheme are based essentially on three steps. First a copy of a stable state of the concurrent object is made; then the service is applied to that copy; and finally, if the object has not changed since the instant of the copy, than an atomic substitution of the object's state with the modified copy is performed. This process is repeated until it succeeds.

The separation of “reader” and “writer” services is also advantageous for the implementation of this scheme. “Reader” services do not require an atomic substitution of the object's state, hence its implementation is much lighter.

Figures 5 and 6 present generic algorithms (in a pseudo-C++ language) for “writer” and “reader” services. In both

¹²To be more precise, as it will be seen ahead, Herlihy demonstrates the existence of universal algorithms to shared data structures, not necessarily to concurrent objects.

cases, a copy of `obj` is made to `obj_cpy` (3.), after which, if the copy has succeeded (4.) then the service is executed using the object's copy (5.). In the case of "writers", and if linearizability is verified, the state of `obj` is replaced with the state of `obj_cpy` (6.). If this replacement fails then all the process is repeated (7.).

Hence to implement this scheme with those simplified generic algorithms, it is necessary that, in the concurrent object, the following four conditions hold:

1. It has to be possible anytime, to make copies of the object's state (regardless of it being in a stable time or not);
2. There has to be a way to detect the existence of modifications in the object's state since the instant in which the copy was made (so that it becomes possible to verify if a stable copy was made, and also to implement a linearizable behavior);
3. It has to be possible to atomically replace the object's state by a copy;
4. It must be possible to repeat the execution of services without affecting their external behavior nor the object's correctness.

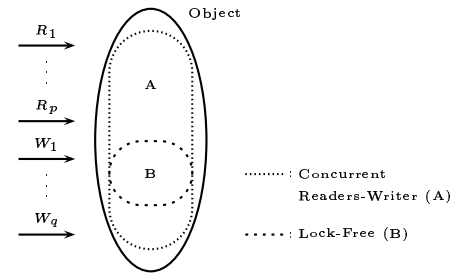
Taking into consideration the several proposals found in the bibliography to implement this synchronization scheme [14, 15, 11], from all those four conditions it is the last one which poses more restrictions to its static safe realizability. Even considering that the execution of services is made on a separate object's copy, not all services are able to be repeatedly executed without side-effects. For example, a service which invokes a writing routine to an external terminal device (or for that matter: to any external file), or the reading of information from external users, in general are not, repeatable.

On the other side, the services which only modify the value of attributes can be made repeatable, for example, with the algorithms presented here.

Service reversibility

A service is repeatable if its eventual effect in the system's state – program or eventual external entities depending on it – resulting from its execution, is reversible to the exact system's state that existed immediately before the execution began.

So, in order for a service to be reversible, its algorithm has also to be reversible. In imperative languages, for an algorithm to be reversible it is necessary and sufficient to ensure that all of its instructions are also reversible. The normal conditional, repetitive, and assignment instructions can all be reversible as long as they apply to the separated object's state copy. Only in the execution of other services in the algorithm, can there be some reversibility problems, if those services are not reversible themselves. For example, services



$$COA_A = COA_{CRW} \quad COA_B = COA_{LF}$$

Figure 7: Example of a mixed synchronization scheme

that interact with entities external to program, are not reversible (as happens in the external input/output routines), unless they only observe the state of those entities.

Hence, this synchronization scheme is only statically realizable in a safe way if the compiling system is able to correctly identify all the reversible services of each concurrent object.

Table 1 summarizes the most important requirements posed on the compiling system by the presented synchronization schemes.

4.6 Mixed Synchronization Schemes

Another interesting approach is to consider the possibility of using different synchronization schemes, simultaneously or alternating in time, within a concurrent object. Such mixed combinations of synchronism would have, of course, to observe all the required correctness conditions including the necessity of total object covering.

4.6.1 Mixed Exclusion Schemes

One possible way of combining several synchronization schemes is to impose mutual exclusion between them. For example, an object can possess a group of services which could, within themselves, be synchronized by a lock-free scheme, and others that, due of not being reversible, require mutual exclusion, readers-writer exclusion or concurrent readers-writer (figure 7) schemes with all of the object's services. In this situation it would be perfectly safe to use an asynchronous group mutual exclusion mechanism [21]. Using this mechanism, several processors can concurrently access the lock-free services, but in mutual exclusion with the remaining processors attempting to execute other services.

Another situation with a similar solution occurs when we are interested in having different synchronization schemes depending on the context in which the object is used. For example, in MP-EIFFEL an object can be reserved to be used exclusively by a single processor for a sequence of calls to its services [6]. If that object happens to have, for instance, a lock-free synchronization scheme, then both uses would not be possible, limiting the usability of more powerful synchronization schemes. A solution to this problem is to use a mixed scheme with both synchronizations, implemented with a group mutual exclusion mechanism to prevent the simultaneous use of both schemes. In this way, we are able to safely switch, at run time, between different synchronization

Table 1: Non-mixed synchronization schemes requirements

	Monitors	Readers-Writer Exclusion	Concurrent Readers-Writer	Lock-Free
Concurrent object identification	Yes	Yes	Yes	Yes
Side-effect identification	No	Yes	Yes	Yes
Reversible "readers"	No	No	Yes	Yes
Reversible "readers" and "writers"	No	No	No	Yes

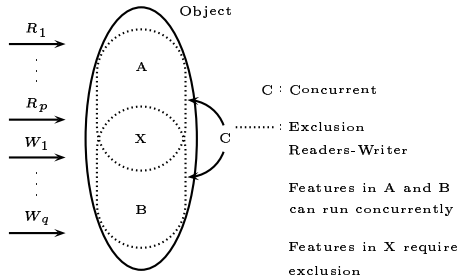


Figure 8: Double readers-writer exclusion

schemes in the same object, making full use of less restrictive schemes.

Correctness of Mixed Exclusion Schemes

It is safe to use any combination of mixed exclusion schemes if the following conditions are observed:

- a) Total object covering;
- b) Each of the synchronization schemes are safe within the the part of the object it applies (which is a subset of all the object's services).

The demonstration of this correctness condition is straightforward. Since the mechanism of group mutual exclusion, by definition, ensures that at most only one of the synchronization groups is active, and being also ensured that all of the object's services are synchronized by at least one group (there could be more than one), it is easy to conclude that it is sufficient to make sure that each group of synchronization schemes is safe in the subset of services to which it applies.

4.6.2 Mixed Concurrent Schemes

By definition, the vast majority of mixed concurrent combinations schemes is not safe. A concurrent modification of concurrent object attributes leads almost always to race conditions in their access from which can result, in an unpredictable way, senseless incorrect values for those attributes, breaking the class's invariant.

However, in certain particular cases it looks like it could make sense to allow, in a very disciplined way, the concurrent access to the object, even without requiring lock-free or readers-writer concurrent synchronization schemes. For example, the use of two or more concurrent groups of mutual or readers-writer exclusion, (figure 8) within an object – each one protecting the access to a separate group of attributes – not being in general safe since nothing ensures

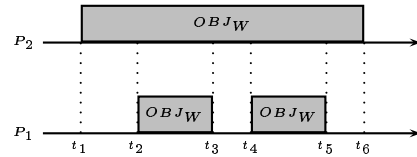


Figure 9: Wrong execution in an object with mixed concurrent synchronization

that in such a situation the invariant will hold when tested, can be sequentially consistent if some restrictions are imposed.

Using a real life example analogy, if we have a CAR object it would be safe to concurrently replace a tire and change its oil, without necessarily having to impose a lock-free scheme (that is, without the necessity of imposing the reversibility of either of those operations).

In order to allow that type of schemes it is necessary to ensure, at least, its sequential consistency, without forgetting the essential verification of the class invariant, and the service preconditions and postconditions. First lets take a closer look to the correctness requirement of an object's service S [29, pages 368–370]:

$$\{INV \text{ and } PRE_S\} SERVICE - BODY_S \{INV \text{ and } POST_S\}$$

So the execution of an object service is correct if, before its execution, the class invariant and the service precondition are true, and, after the execution, the class invariant is again verified together with the service postcondition.

Assuming, for the sake of the argument, only calls to services which may change the object's state ("writers"), the execution presented in figure 9 is not sequentially consistent, since the processor P_1 cannot safely test the class's invariant in the interval $[t_3, t_4]$ between its calls to object's services.

Sequentially consistent invariant verification

Taking a closer look at the figure 9, several considerations can be drawn. From the point of view of processor P_1 it would be sequentially consistent to advance the invariant verification from the instant t_2 to the instant t_1 , since if the invariant holds in t_1 it would also hold in t_2 if there wasn't the interference of processor P_2 . So, it would be perfectly acceptable to reuse the invariant test done by P_2 in t_1 , to the processor P_1 (meaning to assume the invariant of t_1 in t_2).

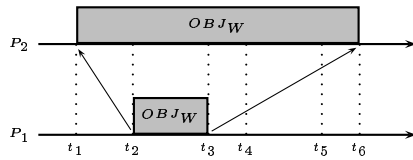


Figure 10: Correct execution in an object with mixed concurrent synchronization

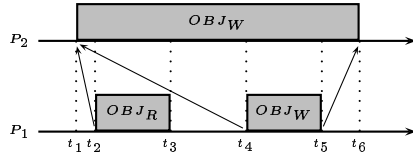


Figure 11: Correct execution in an object with mixed concurrent synchronization

In a similar way, it would be sequentially consistent to delay and reuse the invariant test done by P_1 in t_3 to P_2 in t_6 , if, meanwhile, no more calls to object services on behalf of P_1 are allowed (figure 10)¹³.

On the other hand, the situation presented in figure 11, although it involves two service executions by processor P_1 concurrently with one execution of P_2 , can be considered safe, since the invariant cannot change during “reader” service calls, which is why, the invariant verified in the instant t_1 can be consistently reused in instants t_2 , t_3 and t_4 .

The case presented in figure 12 is not correct since when the processor P_1 begins a “reader” service execution in t_4 , it is not possible to reuse nor to verify the class’s invariant.

To complete the analysis to this type of synchronization schemes, there are two situations that need to be taken care of. The first one occurs when the first concurrent execution is done by a “reader” service. In this case, it is easy to conclude that the invariant, from the point of view of the processor executing that service, will be the same at the end

¹³This behavior affects exception handling, but this problem dealt in MP-EIFFEL goes beyond the scope of this article.

¹⁴Side-effect free.

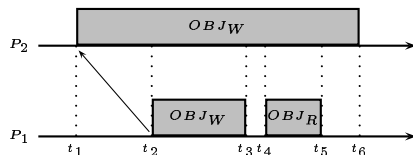


Figure 12: Wrong execution in an object with mixed concurrent synchronization

of the execution. Hence, the execution of this type of services is irrelevant to the correctness of the mixed concurrent synchronization schemes, and so, can be ignored.

Finally, the first “writer” service entering a concurrent execution zone, need not to be the last “writer” to leave (as happens in the figures shown). What needs to be imposed is that the “input” invariant to be reused, will be the one in the beginning of the execution of the first “writer”, and that the “output” invariant to be the one occurring at the end of the last “writer”.

Generalizing all those cases:

Concurrent Verification of Invariants

In a concurrent execution of several processors within an object in the presence of mixed concurrent synchronization schemes, it is sequentially consistent to verify the invariant only when the first processor begins, and the last processor finishes, the execution of “writer” services on the object, if in that interval, the following conditions hold:

- Any processor may execute all the “reader” services it wants, as long as all of them precede a possible invocation of a “writer” service;
- Each processor may only invoke a single “writer” service;
- After the execution of a “writer” service, a processor may not execute any other service.

Getting back to the CAR example, with a mixed concurrent scheme with multiple readers-writer exclusion groups respecting the above conditions, it would be possible to concurrently replace a tire and change its oil by different employees (processors), but restricting each employee to only perform one operation concurrently with the other employees. That is, an employee can only proceed its work on the car if it is ensured that the last one was done correctly, thus not compromising the car’s invariant. It is not hard to conclude that these considerations are generalized to the concurrent mixture of other types of synchronization schemes.

Correctness of Mixed Concurrent Schemes

It is safe to concurrently mix two or more synchronization schemes as long as the following conditions are observed:

- Total object covering;
- Each synchronization scheme protects a separate group of object’s attributes;
- The criterion for concurrent verification of invariants is observed.

4.6.3 Realizability

One interesting characteristic of mixed schemes is the fact that the requirements posed by each scheme don't need to apply to the whole object, but only to a subset of its services.

The implementation of **mixed exclusion schemes** requires that each scheme is implemented only to the subset of services to which they apply (in some cases, it can be the whole object), and, as already mentioned, the use of the mechanism of asynchronous group mutual exclusion [21].

In the case of **mixed concurrent schemes**, the compiling system needs to gather more information about the program. In particular, it needs to know the subset of the object's attributes used, directly or indirectly, by each one of the services of concurrent objects. This information is essential in order for a static correctness verification of the application of this scheme. Only services that never interfere with each other may execute concurrently. All of the remaining services are required to execute in mutual or reader-writer exclusion with the services using the same attributes.

To implement a **mixed concurrent scheme** synchronization algorithm, it is sufficient the use of a simple approach based in a shared atomic counter. The appendix A has a possible safe implementation in the language C¹⁵ of this algorithm for the particular situation in which processors are POSIX-THREADS. In this implementation, all the necessary synchronization is done in the invariant verification, so "reader" and "writer" services only need to call the appropriate invariant testing functions. For "writer" services, each executing processor¹⁶ not only reuses the invariant test done by the first processor, but also finishes its execution only when the last processor terminates the invariant verification. This ensures that at most, each processor only executes one "writer" service, and also that no "reader" execution is done afterwards. "Reader" services reuse the last tested invariant done after a "writer" service without blocking.

4.7 Local Processor Attributes

So far we have been assuming that a concurrent object has only shared attributes. That is no doubt what is usually desired when concurrent objects are to be implemented. However, it need not to be the case in all situations. For example, if one wants to implement internally in an object a caching structure, in order to reuse the object's history to enhance its performance in the future, but without compromising the concurrent safety properties of "reader" services, it would be interesting to have some kind of processor attributes free from intra-processor interference.

In the *POSIX-THREADS* library [3] there are the named "thread-specific data" which serve a similar purpose¹⁷.

The basic idea of this mechanism is very simple. Knowing that, in concurrent objects, intra-processor interference results from the use of shared attributes, why not allow –

¹⁵Used sometimes in the code generation phase of compiling systems of many languages.

¹⁶Implemented only as POSIX-THREADS.

¹⁷Although local processor attributes even when processors are simply threads, need not and should not, be implemented with thread-specific data.

in the rare, but important, cases where it may be useful – objects to have attributes local to each processor (with the same syntactic signature)?

Services which only change this type of attributes are – from the point of view of intra-object concurrency – equivalent to "reader" services, except that they might not be automatically reversible. In order to make them reversible it is necessary to be able to backtrack all of the local processor attributes to the values they had at the beginning of the routine.

5. SYNCHRONISM SCHEME SELECTION

An important question remains to be answered. Although there are several possibilities for static safe implementations of many synchronization schemes, who should choose which to used?

A common approach that is not acceptable for static safe concurrent systems is to leave both the choice and the programming of synchronization scheme in the programmers hands. However, this approach is unsafe.

There are several safe alternatives:

1. The programming language system predefines a single scheme;
2. The programming language system allows the programmer to choose a desired safe scheme;
3. The compiling system chooses the appropriate schemes using some heuristics;
4. A mixture of all of them.

The first case is the most frequent in static concurrent languages such as ADA95 or SCOOP. It has the advantage of being simple, but it is prone to systemic safety problems such as deadlocks¹⁸, and is also too inflexible.

The second possibility – assuming that the compiling system has the ability to verify and safely implement the chosen synchronization schemes – is a much better option.

In the third approach the language defines a simple and clear semantics for concurrent objects, and the compiling system ensures their observance, being free to choose any safe synchronization scheme. However, at first sight it seems that this approach can pose some complexity problems, since it is hard to imagine a deterministic algorithm that, given a concurrent program and an execution context, would be able to choose the most appropriate scheme. Different concerns, such as for example: efficiency, memory usage, deadlock prevention, may collide with each other (hence the use of the heuristics term). Also, different application areas, such as hard real-time systems, pose different demands on the synchronization schemes.

Finally, another possibility is to try using the best of each of the previous approaches. The programming language de-

¹⁸Assuming, of course, the usual choice of a blocking scheme.

finer clear semantics for concurrency constructs. In a different level (Concurrency Control Language in MP-EIFFEL), there is the possibility for the programmer to choose different synchronization schemes, although the compiling system may reject them, if it is unable to implement them safely. The compiling system, for example, in the presence of possible liveness problems, may attempt to solve them by choosing appropriate lock-free synchronization schemes. This is the choice made in MP-EIFFEL, although, at the moment, there isn't much practical experience (hence same caution needs to be taken regarding its ability to properly handle those choices).

6. FINAL REMARKS

Although the work presented in this article is integrated in the implementation of MP-EIFFEL, many of its results are applicable in other programming languages.

As mentioned throughout the paper, some concurrency safety problems are yet to be solved. That is the case of those arising from liveness issues. The possibility of having safe static implementation of synchronization schemes with higher *COA* values, may decisively contribute towards their resolution.

7. REFERENCES

- [1] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.
- [2] J. H. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 111–122, Dec. 1997.
- [3] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.
- [4] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.
- [5] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Communications of the ACM*, 14(10):667–668, 1971.
- [6] M. O. e Silva. Concurrent object-oriented programming: The mp-eiffel approach. *Journal of Object Technology: Special issue: TOOLS USA 2003*, 3(4):97–124, April 2004.
- [7] J. Guttag. Abstract data types and the development of data structures. *Commun. ACM*, 20(6):396–404, 1977.
- [8] P. B. Hansen. The purpose of concurrent pascal. In *Proceedings of the international conference on Reliable software*, pages 305–309, 1975.
- [9] P. B. Hansen. Monitors and concurrent pascal: a personal history. In *The second ACM SIGPLAN conference on History of programming languages*, pages 1–35. ACM Press, 1993.
- [10] P. B. Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, 1999.
- [11] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 388–402. ACM Press, 2003.
- [12] M. Herlihy. A methodology for implementing highly concurrent data structures. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 197–206. ACM Press, 1990.
- [13] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.
- [14] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.
- [15] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003.
- [16] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26. ACM Press, 1987.
- [17] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [18] C. A. R. Hoare. Hints on programming language design. Technical Report STAN-CS-73-403, Stanford Artificial Intelligence Laboratory, Computer Science Department, Stanford University, 1973.
- [19] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
- [20] J. T. Joseph A. Goguen and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, Englewood Cliffs (N.J.), 1978.
- [21] Y.-J. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.
- [22] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.
- [23] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
- [24] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

- [25] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, 1974.
- [26] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
- [27] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.
- [28] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, 1993.
- [29] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
- [30] G. L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.

APPENDIX

A. MIXED CONCURRENT SCHEMES

A.1 Invariant testing implementation

```
#include <pthread.h>

typedef struct
{
    int counter;
    int done_start;
    int Result_start;
    int Result_end;
    pthread_mutex_t mtx;
    pthread_cond_t cnd;
} INVARIANT_SYNC;
#define INVARIANT_SYNC_INIT \
    {0,0,0,0,PTHREAD_MUTEX_INITIALIZER,PTHREAD_COND_INITIALIZER}

int command_test_invariant(int (*inv)(void *obj),void *obj,
                          INVARIANT_SYNC *synch,int start_of_routine)
{
    int Result;

    pthread_mutex_lock(&synch->mtx);
    if (start_of_routine)
    {
        synch->counter++;
        if (!synch->done_start)
        {
            // Invariant checked only in the first routine
            // (except for creation command, instead of rechecking
            // the invariant, we could reuse the last Result_end).
            synch->Result_start = (*inv)(obj);
            synch->done_start = 1;
        }
        // Invariant result reused for all concurrent routines
        Result = synch->Result_start;
    }
    else // end_of_routine
    {
        synch->counter--;
        if (synch->counter == 0)
        {
            // Invariant checked only in the last routine
            synch->done_start = 0;
            synch->Result_end = (*inv)(obj);
            // awake all waiting processors (barrier end)
            pthread_cond_broadcast(&synch->cnd);
        }
        else
        {
            // wait for the last routine
            while(synch->counter > 0)
                pthread_cond_wait(&synch->cnd,&synch->mtx);
        }
        Result = synch->Result_end;
    }
    pthread_mutex_unlock(&synch->mtx);

    return Result;
}

int query_test_invariant(int (*inv)(void *obj),void *obj,
                        INVARIANT_SYNC *synch)
```

```
{
    int Result;

    pthread_mutex_lock(&synch->mtx);
    // fetch last invariant verification
    if (synch->done_start)
        Result = synch->Result_start;
    else
        Result = synch->Result_end;
    pthread_mutex_unlock(&synch->mtx);

    return Result;
}
```

A.2 “Reader” services implementation

```
1. if (!query_test_invariant(...))
1.1. raise_invariant_exception(...);
2. if (!test_precondition(...))
2.1. raise_precondition_exception(...);
3. Result = execute_query_body(...);
4. if (!test_postcondition(...))
4.1. raise_postcondition_exception(...);
5. if (!query_test_invariant(...))
5.1. raise_invariant_exception(...);
```

A.3 “Writer” services implementation

```
1. if (!command_test_invariant(...,1))
1.1. raise_invariant_exception(...);
2. if (!test_precondition(...))
2.1. raise_precondition_exception(...);
3. execute_command_body(...);
4. if (!test_postcondition(...))
4.1. raise_postcondition_exception(...);
5. if (!command_test_invariant(...,0))
5.1. raise_invariant_exception(...);
```