Automatic Realizations of Statically Safe Intra-Object Synchronization Schemes in MP-Eiffel

Miguel Oliveira e Silva

IEETA—DETI, Universidade de Aveiro, Aveiro, Portugal,
mos@det.ua.pt,
WWW home page: http://www.ieeta.pt/~mos

Abstract. This article presents the approach taken in MP-EIFFEL to handle intra-object synchronization of concurrent objects. The correctness of concurrent objects is discussed assuming the existence of contract language mechanisms. Several synchronization schemes are presented and their automatic realization by the compiling system is discussed. A new proposal for automatic realizations of mixed synchronization schemes, either in exclusion or in concurrency, is presented and discussed. It is shown that one of those mixed schemes provides a solution for the safe integration of intra-object and inter-objects synchronization schemes within a concurrent object. The problem of conditional synchronization is also considered.

1 Introduction

MP-EIFFEL[1] [1] is a prototype language which is being designed and implemented to test and validate concurrent object-oriented language mechanisms. Its goal is to provide a coherent group of expressive and safe concurrent language constructs suitable for general concurrent programming based on static typed systems, contracts, and pure[2] object-oriented languages.

Instead of developing from scratch a new object-oriented language, we have decided to base our work upon an appropriate existing language. The choice was (obviously) EIFFEL [2] due to its simplicity, elegance, static type system, and because it is one of the (very) few languages with Design-by-Contract^{TM} (DbC) mechanisms [2, Chap. 9]. DbC mechanisms are very important not only to assert the correctness of (sequential) objects (and all their uses) [3, Chap. 11], but also to describe their semantics as instances of possibly partial Abstract Data Type (ADT) implementations [3, Chap. 6]. In concurrent programs it would be desirable to ensure that the same properties also apply to concurrent objects[3].

[1] Multi-Processor EIFFEL.

[2] An object-oriented language is considered pure if its programs are only composed of communicating objects.

[3] A precise definition of concurrent objects will be presented in Sect. 2.4.

The first goal of this article will be to provide a definition of concurrent object correctness, taking into consideration not only its possible concurrent uses (which is a well established theory) but also the DbC mechanisms. The relation between (isolated) concurrent objects and class contracts is also established. Of particular interest will be the relation of concurrent contracts with the correctness and behavior of the (concurrent) object.

The second goal of this article is to assess the realizability of (internal) object synchronization schemes by the compiling system. We describe several synchronization schemes, identify the information that the compiling system must gather in order to allow their automatic realization, and present some algorithms used to implement them. The ability to automatically synchronize concurrent objects using different schemes enables the approach, recently adopted in MP-EIFFEL[4], of detaching concurrent entity program annotation from specific synchronization schemes (unlike other object-oriented concurrent languages). As a result, MP-EIFFEL objects might be tuned, after program development, to many different synchronization schemes in order to improve some of its properties such as liveness or the availability of concurrent objects.

A relevant contribution is the study done on the automatic realization of mixed synchronization schemes. In particular, one such scheme is originally proposed to solve the problem of integrating two different synchronization types in a concurrent object. One resulting from internal object safety concerns (intra-object synchronization), and the other from external client needs (inter-object synchronization).

2 Basic Definitions

For a better comprehension of this article, some basic terms and definitions will be presented here.

2.1 Service, Attribute, Command and Query

An object is composed by a set of **services**, which may be **attributes** (object data fields), or **routines** (object methods). When necessary, services might also be classified as **queries** and **commands**. Queries are services which return object properties (they are used to observe the object). Commands are services which may modify the object's state. Queries that never modify the object's state are called **pure**. Usually they are functions, although they can also be attributes. Commands are procedures (routines without return values, or *void* functions in languages that use C terminology).

In the context of object attachment, we will use the term **entity** to refer to identifiers in the class text that might. at run-time, become attached to objects. In EIFFEL there are four types of entities [2, page 275]: attributes, local entities of routines, formal routine arguments, and the reference to the current object (**Current** in EIFFEL and **this** in JAVA [4] and C++ [5]).

[4] In a previous version [1], concurrent objects always used a readers-writer exclusion mechanism.

2.2 Processor

It is common to use the term *thread* to represent program execution units which operate on a shared memory environment in the same computer. Likewise, *process* is frequently used for more decoupled execution units (in UNIX a process can share its address space with multiple threads). Since those terms are usually connected with specific operating system execution units, and because we are interested in abstract concurrent programming in which each execution entity may be implemented in many different ways (even through different processes in several computers on a distributed network), we will use the abstract notion of **processor** adapted from Meyer [3, page 964]:

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

A processor might be implemented as a process, a thread, a group of processes in a network of distributed computers, or in any other realizable way.

In the context of the execution of services in objects, a processor will be called **writer** if it is executing a service that might change the object's state (usually commands); and it will be called **reader** if it is executing side-effect free services (pure queries).

2.3 Models of Inter-Processor Communication

There are two basic models for inter-processor[5] communication: message passing (direct)[6]; and shared memory (indirect).

In message passing inter-processor communication, by definition, the sender (caller) processor will always be different from the receiver (callee) processor. The ACTOR family of languages [6] and SCOOP[7] [7, 3] restrict their concurrent communication mechanisms to this model (ADA95 [8] also has a message passing mechanism named rendezvous). In the original proposal of SCOOP each concurrent object is handled by a single processor throughout its entire life (which makes its synchronization very simple).

In shared memory communication the caller processor is the same as the callee processor. Concurrent programming systems that use this communication model are the POSIX-THREADS [9] library, ADA95 protected types, and the language JAVA [4].

MP-EIFFEL has language constructs for both types of communication models. Message passing is expressed by triggers and remote entities, and shared memory through shared and remote entities (a detailed description of these mechanisms can be found in [1])[8].

---

[5]

[6] Not to be confused with object-oriented inter-object message passing communication. Both share the message passing communication model semantics, but applied to different communicating parties.

[7] Simple Concurrent Object-Oriented Programming.

[8] Section 2.4 briefly defines shared and remote entities.

2.4 Concurrent Objects

In a concurrent object-oriented program it is essential to identify all the objects – named concurrent – that (might) require a proper synchronization scheme in order to be correctly and safely used. Therefore, a concurrent object will be an object whose services might be requested by more than one processor in overlapping times, or in which the (direct) caller and callee processors might be different. All objects that are not concurrent are named sequential.

In the case of inter-processor message passing communication mechanisms all the objects able to directly handle requests from different processors are concurrent. In SCOOP, the static typed system is used to conservatively identify concurrent objects. A concurrent annotation (**separate**) is used to identify all the entities to which concurrent objects might be attached. Its properties [3, pages 973–975] ensure that concurrent objects cannot be unsafely attached to sequential entities and sequential objects to concurrent entities.

The identification of concurrent objects when using shared memory inter-processor communication mechanisms is, in general, much more difficult. One possibility (which could also be applied to the other communication model) is to delegate such responsibility on the programmer, as happens, for example, in JAVA. We are not interested in such error prone unsafe approach to concurrency. Instead we want to statically identify (even if conservatively) all possible concurrent objects, and, using such knowledge, to automatically synchronize those objects ensuring their correctness and safety. Like SCOOP, MP-EIFFEL uses the static typed system to unambiguously identify all possible concurrent objects. To that goal two new type annotations were added to the EIFFEL type system: **shared** and **remote** (entities with one of these annotations are called concurrent entities). Shared objects are concurrent objects that can be observed and modified by different processors. Remote objects are also concurrent objects with the restriction that there is only one "writer" processor (a unique processor is allowed to modify its state). The language type rules [1] ensure the safe use of concurrent and sequential objects.

2.5 Intra-Object and Inter-Object Synchronization

A concurrent object might be required to meet different synchronization needs. On one hand, an object is required to protect itself from concurrent executions of its services. This type of synchronization is named **intra-object synchronization**.

On the other hand, clients might require the exclusive use of concurrent objects throughout the execution of more than one of their services. This synchronization type, which results from external uses of the object, is named **inter-object synchronization**.

In concurrent objects in which both synchronization types are required, it is necessary to ensure that at least one general solution (automatically implementable) exists, which allows their correct integration regardless of the intra-object synchronization scheme used. Such solution exists, and will be presented in Sect. 4.5.

2.6 Conditional Synchronization

A service from a concurrent object may not always be available to be used by clients. Often, services are usable only if certain conditions on the object's observable state are met. For example, a request for a `pop` service on a concurrent `STACK` only make sense if the stack is not empty. A possible solution to this problem is to use a **conditional synchronization** scheme in which the client is required to wait until the condition is met (wait condition).

2.7 Concurrent Assertions

An assertion is said to be concurrent if it contains at least one concurrent assertion clause[9]. A concurrent assertion clause is one containing a concurrent boolean condition. Finally, a concurrent boolean condition is a boolean condition with a value that, in the context in which it is to be evaluated, may depend on the behavior of at least one processor other than the one testing it.

Assertions can be decomposed into two sets: a set of sequential assertion clauses, and another of concurrent assertion clauses.

2.8 Concurrent Object Availability

In order to compare different intra-object synchronization schemes, it is useful to have some kind of objective metric 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 sharing some property x operating in an object OBJ (for example, reading or writing properties), and that N_c is the maximum number of such processors which can safely operate concurrently inside OBJ (of course: $N_c <= N_x$), we define the Concurrent Object Availability of OBJ in relation to the processors with the x property as being:

$$COA_x = \frac{N_c}{N_x} \tag{1}$$

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 Concurrent Object Correctness

Object-Oriented software construction is defined by Meyer [3, page 147] as the building of software systems as structured collections of possibly partial abstract

[9] An assertion clause is a simple boolean condition declared inside an assertion.

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 occur between them. A necessary condition for a concurrent object to be correct is that its ADT is never compromised by its concurrent use.

Viewing objects as instances of ADT implementations simplifies their use, and provides a solid theoretical basis for object-oriented programming. It should be noted that objects – being instances of ADT implementations – should include the ADT semantic properties. Those properties can be expressed by class invariants, and service preconditions and postconditions. Unfortunately few languages allow the implementation of these semantically rich ADTs, by including mechanisms to express, and when possible, to test those assertions. EIFFEL is one such rare language (it pioneered this ADT view of objects, promoting the Design by Contract programming methodology), but several other tools are beginning to appear, for example, in JAVA [10] and C++ [11].

In sequential programs, in order not to compromise the correctness and simplicity of ADTs, objects can be externally used only at their stable times [3, page 364]. Such a behavior is relatively easy to ensure in sequential programming, because there is only one processor. However, in concurrent programs in which there is the possibility of intra-object concurrency the problem is much more complex. In the presence of invariants, stable times can only be enforced within the class boundaries if it is forbidden the existence of public modifiable attributes (otherwise, any client could change the object's state, possibly breaking the invariant, outside of its control).

3.1 Linearizability

A sufficient condition to ensure the correctness of concurrent objects is linearizability [12, 13].

Linearizability

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

3.2 Class Contracts

In contract aware languages, linearizability is required to take also into consideration possible executable class assertions which are also applicable to concurrent objects.

A sufficient condition to ensure the correctness of concurrent objects with class assertions, is to consider the (possible) execution of all the applicable assertions as being part of the linearizable object operation.

3.3 Concurrent Contracts

The existence of concurrent objects raise another very interesting problem: besides normal sequential assertions, class contracts can instead make use of concurrent assertions.

In such cases, what should be its correct behavior?

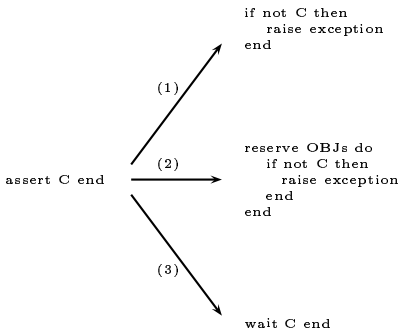

Fig. 1. Possible behaviors of concurrent assertions

Since, by definition, a concurrent assertion depends, at least, on a processor other than the one which is testing the assertion, its normal unsynchronized sequential behavior ((1) in Fig. 1) could clearly create race conditions, hence it is an unacceptable behavior.

Another possibility would be to unconditionally grab all the concurrent objects involved in the assertion (2), and then use the assertion as if it was sequential. This behavior is also a source of race conditions because once the object is reserved for the exclusive use of the processor responsible for testing the assertion, the concurrent assertion might be false depending on unpredictable timing relations between processors.

So, it seems that the only safe behavior is to attach concurrent assertions to wait conditions (conditional synchronization) (3): a concurrent assertion causes its executing processor to wait until the concurrent objects are available and the concurrent assertion is verified. Not surprisingly, this is exactly the same behavior that is proposed for concurrent preconditions in SCOOP.

The verification of any assertion is always of the responsibility of the program code that is (or could be) executed before the assertion. Hence, preconditions are the responsibility of object clients, and invariants and postconditions on the object itself. In a sequential program there is only one processor, so if an assertion is proved (usually by testing it) to be false, then we are clearly in the presence of a programming error, because its value will remain false unless, afterwards, the processor itself does appropriate actions to change it. However, concurrent programs have more than one processor. So if the assertions is concurrent then its state can change without the participation of the processor which is doing

its runtime verification. So again, the only safe behavior will be to ensure that concurrent assertions behave as wait conditions.

In the case of intra-object synchronization, the only concurrent assertions that are relevant are preconditions and (eventually) invariants (when tested before service execution).

Concurrent contracts (and also concurrent conditions within structured conditional and iterative instructions) are also of key importance for inter-object synchronization, but such discussion is beyond the scope of this article (a draft article on inter-object synchronization in MP-EIFFEL can be found in [14]). Also beyond the scope of this article is the possible dynamic nature of concurrent assertions. The same assertion, depending on the context of its verification, might behave concurrently or sequentially (see [14]).

3.4 Total Object Covering

A trivial necessary condition regarding intra-object synchronization 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 an appropriate synchronization mechanism.

One of the strongest objections of Brinch Hansen [15] to the concurrent mechanisms of the language JAVA is the fact that its programs may not observe this rule, posing safety problems.

4 Intra-Object Synchronization Schemes

Having defined the essential correctness conditions and requirements to observe in the synchronization of concurrent objects, in this section we will present several synchronization schemes with sufficient realizability conditions to allow an automatic safe implementation by the compiling system.

4.1 Monitors

A simple and sufficient approach to ensure linearizability, is to consider each object to be a monitor (Fig. 2). Interestingly, Hoare [16] and Brinch Hansen [17] themselves have recognized the importance of the class concept of the first object-oriented language, SIMULA, when they proposed the monitor synchronization scheme.

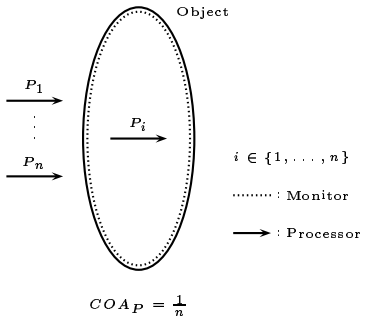

Fig. 2. Monitors

Monitors are the simplest synchronization scheme. The price to pay for that simplicity is that objects 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.

The concurrency mechanisms of JAVA were initially designed to be approximations of monitors [18, page 399], but their intents have failed in some important aspects [15]. The current version of the language [4], although not solving some of the original monitor problems, allows, although to a limited extent, the use of other synchronizing schemes besides monitors [19].

Realizability Monitors pose relatively few requirements on the compiling system. A trivial requirement is the necessity for identifying all of the object's public services. Those services will need to be protected with monitor synchronization code.

A sufficient algorithm to implement this synchronization behavior, is to create a proxy class with an identical interface of the unsynchronized class, in which the monitor synchronization code is implemented. This approach has also the advantage of avoiding the over-synchronization problem of calling public services inside the object.

Monitors implement conditional synchronization through the use of condition variables. A possible (though inefficient) algorithm would be to attach a single condition variable to the monitor (object) and to signal all waiting processors (broadcast) at the end of the object's public routines. It would also be necessary to convert each concurrent precondition (and invariant) to a wait instruction and relevant code on the condition variable.

As an example, appendix A.2 presents a possible automatic implementation of a monitor synchronization scheme of a stack class (described in A.1). As is easy to verify, the automatic translation of the stack class (into `MONITOR_STACK`) requires little semantic knowledge on the part of the compiling system. Although the presented translation algorithm takes advantage of the ability to distinguish commands and impure queries from pure queries; it is not a monitor requirement but simply an optimization of the conditional synchronization mechanism.

Section 5 discusses other more efficient possibilities to optimize conditional synchronization algorithms.

4.2 Readers-Writer Exclusion

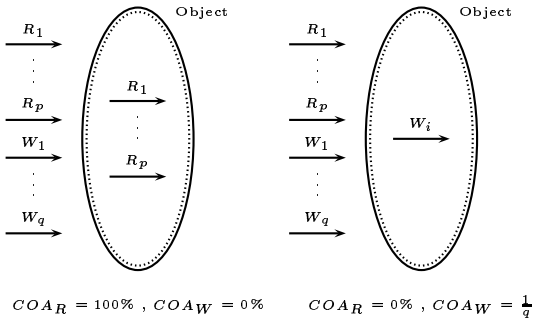

Fig. 3. Readers-Writer Exclusion

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 commands) is being processed, allowing the concurrent processing of the remaining (pure query) services.

Hence an approach using the synchronization scheme of readers-writer exclusion [20] (a writer processor excludes all the others, but multiple reader processors can operate concurrently) is also a valid and safe choice (Fig. 3). 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) liveness problems such as deadlocks.

This synchronization scheme is used in the language Ada95 (protected types), and was also the first approach taken by the prototype language MP-Eiffel, proposed by the author.

Realizability This synchronization scheme is "better" (higher average COA values) than monitors, but the compiling system requires a little more information on the concurrent object class. Unlike monitors, this scheme requires the ability to distinguish commands and impure queries from pure queries.

In MP-Eiffel this is implemented through the following reasoning. A service is considered as having 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. In qualified calls to routines we take the conservative approach of

verifying the purity of all possible dynamic binding routines. Recursive routines (either directly or through other routines), pose no problem to this approach because the compiling system keeps track of the routines already traversed.

Conditional synchronization differs from the monitor case due to the fact that there are two different lock instructions (one for reading and one for writing). Other than that, the algorithm can be quite similar.

Appendix A.3 presents a readers-writer exclusion possible automatic implementation of the stack class.

4.3 Concurrent Readers-Writer

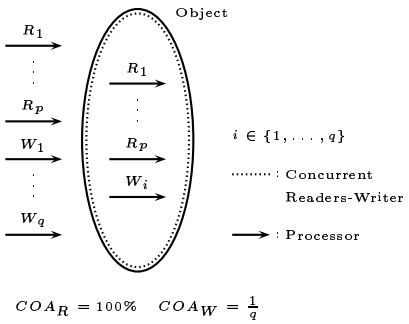

Fig. 4. Concurrent Readers-Writer

Lamport [21] has proposed a generalization to the previous synchronization scheme which allows the concurrent access of multiple reader processors and a single writer processor. Mutual exclusion is only required to multiple writer processors (Fig. 4). 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).

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 scheme. 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 [21, 22].

A possible solution, in some cases, to this problem is proposed by Peterson [22]. 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[10], Peterson [22] proposes a wait free algorithm to any processor which makes the object always available ($COA = 100\%$) to any processor.

Realizability Lamport's 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, it is necessary to foresee the situation in which there might exist assertion failures (invariant, preconditions or others) during the reader execution, resulting from changes on the object state produced by a concurrent writer execution. 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 (in this particular case) race-condition ones.

Conditional synchronization can be similar to the monitor's case.

Appendix A.4 presents a proxy class with an implementation of this algorithm.

4.4 Lock-Free Synchronization

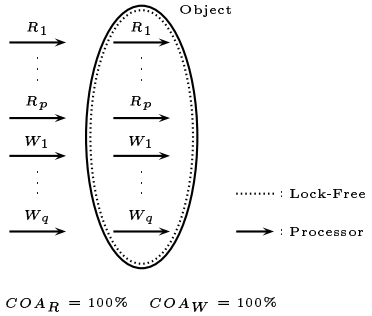

Fig. 5. Lock-Free Synchronization

[10] This situation occurs in MP-EIFFEL when remote entities are used in shared memory communication mechanisms.

A group of synchronization schemes that has been deserving a growing enthusiastic interest are lock-free and wait-free synchronization [23] (Fig. 5). 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[11] (making it immune to deadlocks) and tolerance to faults on other processors. These characteristics make it especially suitable for real-time programming [24].

Currently, this type of synchronism is seldom used, though this situation is expected to change in the future. A good sign towards that direction was the public release of a library of classes for JAVA (JSR 166: Concurrency Utilities [25]) that use this type of synchronism.

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. Here we are interested in a preliminary study on the the possibility for future automatic safe implementations of lock-free synchronization schemes.

Basic Concepts In general, lock-free synchronization algorithms are based on the total, or partial, duplication of the object's attributes and, when necessary, in concentrating all of the necessary modifications to that object in a unique atomic modification. 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 foresee and accept possible failures (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 [26, 23] has demonstrated that there are universal algorithms able to implement this synchronism in concurrent objects observing the linearizability condition, presenting also universal methodologies (though not very efficient) [26, 27] for its implementation. The presented methodology, as mentioned by Herlihy, is adaptable to be automatically executed by the compiling system.

Other possible lock-free related algorithms are based on software transactional memory [28]. Those algorithms work in a analogous way as transactions in database systems. Transactions proceed in three steps. First a transaction is announced, then the executions of the required operations is performed, and finally an attempt to commit the transaction is performed. On failure, it is ensured that the the transaction did not change the memory state. Otherwise, the results of the transaction take (atomic) effect. This transaction process is repeated until it succeeds. Harris and Fraser [29] proposed a language mechanism

[11] Except when conditional synchronization is required.

for JAVA (strongly based on Hoare's conditional critical regions) which takes advantage on the possibilities of software transactional memory for general lock-free algorithms (it also includes a mechanism for conditional synchronization). If the requirements imposed on the compiling system presented ahead are met, Harris and Fraser implementation can be safely used to implement a lock-free synchronization scheme in concurrent objects (in order to do that, it is required that the atomic construct is applied to the whole public services of the object).

Realizability Either Herlihy's generic algorithm [27] or the software transactional memory algorithm, require the ability to take copies of the state of objects, and the necessity of allowing possible repetitions of services. This last requirement, is the one which restricts the most the static safe realizability of these algorithms.

In fact, even taking into consideration that the execution of a service by a processor apply to a separate stable copy of the object local to that processor, not all services can be repeatedly executed without nasty side-effects to other processors (and the system's state). For example, a service which invokes a writing routine to an external terminal device (or for that matter: to any external file), or which reads information from external users, cannot be safely repeated.

On the other hand, services which only modify the values of attributes are repeatable.

Service repetition

> A service is repeatable if its effect in the system's state – program or eventual external entities depending on it – as a result of its execution, is discardable as if the service did not execute.

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

It should be noted that, unlike the previously presented synchronization schemes, lock-free algorithms are not yet integrated and conveniently experimented in MP-EIFFEL (hopefully that situation will change in the near future).

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

Table 1. Compiling system requirements posed by non-mixed schemes

	Monitors	Readers-Writer Exclusion	Concurrent Readers-Writer	Lock-Free
Concurrent object identification	**Yes**	**Yes**	**Yes**	**Yes**
Pure query identification	**No**	**Yes**	**Yes**	**Yes**
Repeatable pure queries identification	**No**	**No**	**Yes**	**Yes**
Repeatable services identification	**No**	**No**	**No**	**Yes**

4.5 Mixed Synchronization Schemes

So far we have been looking into concurrent object's synchronization as if it required a unique uniform synchronization scheme. However, there is no theoretical reason for not considering the possibility of using different synchronization schemes, simultaneously or alternating in time, within a concurrent object; in an attempt, for example, to increase its concurrent availability. As it will be seen, this approach will also provide a safe generic solution for reserving concurrent objects (which is an inter-object synchronization problem) for exclusive uses of its services, without compromising its intra-object synchronization scheme (which could even be lock-free).

Naturally, such mixed combinations of synchronism have to observe all the required correctness conditions, including – in particular – the necessity of total object covering.

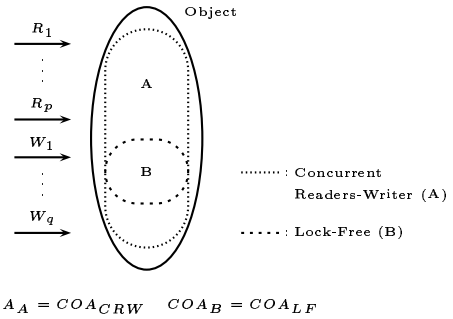

Fig. 6. Example of a mixed synchronization scheme

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 to not being repeatable, require mutual exclusion, readers-writer exclusion or concurrent readers-writer (Fig. 6) schemes with all of the object's services. In this situation it would be perfectly safe to use an asynchronous group mutual exclusion mechanism [30]. 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 externally 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 [1] (inter-object synchronization). 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 schemes in the same object, making full use of less restricting 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.

Mixed Concurrent Schemes By definition, the vast majority of mixed concurrent combinations schemes are 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.

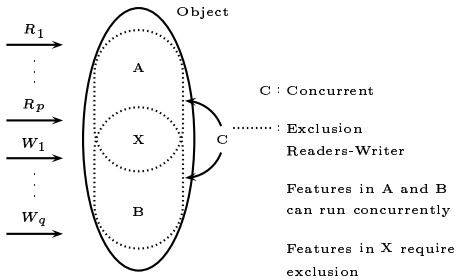

Fig. 7. Double readers-writer exclusion

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, (Fig. 7) within an object – each one protecting the access to a separate group of attributes – not being in general safe since nothing ensures that in such a situation the invariant will hold when tested, can be linearizable 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 repeatability of either of those operations).

Since in this article we take the semantically rich view of objects as being instance of ADT implementations with executable assertions. The implementation of these schemes are required to safely verify all of the object's class assertions. Lets take a closer look to the correctness requirement of an object's service `S` [3, pages 368–370]:

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

So the execution of an object service will be correct if, before its execution, the class invariant and the service precondition are true, and, afterwards, the same happens to the invariant and the service postcondition.

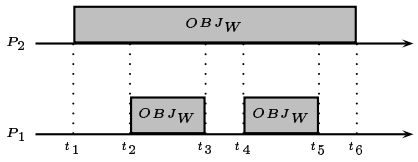

Fig. 8. Wrong execution in an object with mixed concurrent synchronization

Assuming, for the sake of the argument, only calls to writer services (OBJ_W), the execution presented in Fig. 8 is not linearizable, 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.

Linearizable invariant verification

Taking a closer look at the Fig. 8, several considerations can be drawn. From the point of view of processor P_1 it would be linearizable to anticipate 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).

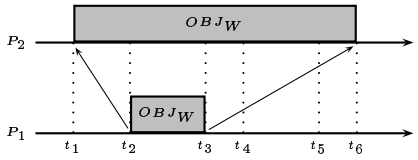

Fig. 9. Correct execution in an object with mixed concurrent synchronization

In a similar way, it would be linearizable 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 (Fig. 9)[12].

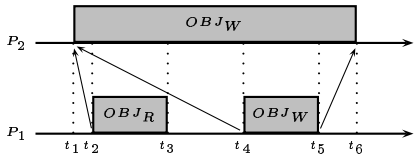

Fig. 10. Correct execution in an object with mixed concurrent synchronization

On the other hand, the situation presented in Fig. 10, 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 (OBJ_R) 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.

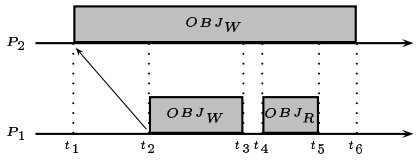

Fig. 11. Wrong execution in an object with mixed concurrent synchronization

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

The case presented in Fig. 11 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 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 linearizable to verify the invariant only when the first writer processor begins, and the last writer processor finishes, if in that interval, the following conditions hold:

a) Any processor may execute all the reader services it wants, as long as all of them precede a possible invocation of a writer service by the same processor;
b) Each processor may only invoke a single writer service;
c) 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:

a) Total object covering;
b) Each synchronization scheme protects a separate group of object's attributes;
c) The criterion for concurrent verification of invariants is observed.

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 [30].

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 B has a possible safe implementation in a C alike language 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 processor[13] 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. It should be noted, that this implementation does not take into consideration conditional synchronization. However, since mixed concurrent schemes do not share attributes, the implementation of conditional synchronization if each mixed scheme, is guaranteed to work within the subset of services to which the scheme applies.

[13] Implemented only as POSIX-THREADS.

5 Conditional Synchronization Optimizations

In Sect. 4 and in appendix A we have presented a simple but inefficient automatic implementation of conditional synchronization. In this section we will discuss several possibilities to optimize such implementations.

A processor may wait until a synchronization condition is verified in two possible ways. Either by continuously testing the condition until it is verified (busy-wait); or it can go to sleep and rely on appropriate awaking signals when the activity of other processors might affect the expected condition.

The first technique might be an acceptable solution if a hardware central processing unit (CPU) is exclusively dedicated to the waiting processor (the CPU would be idle anyway); otherwise this behaviorless waiting process would be a meaningless waste of CPU cycles. We won't consider it here.

The second technique is much more economic use of the CPU resources, but it requires appropriate awaking signals external to that waiting processor.

When the responsibility for implementing correctly this type of synchronization is put on the programmers hands (such as in POSIX-THREADS), the programmer can decide when signaling is required to happen and, most importantly, to whom it should be addressed to. Monitors [16] were designed with this mechanism, in which condition variables were used to distinguish signals (POSIX-THREADS C library implements a similar concept).

In JAVA's approach, the responsibility for testing the synchronization conditions, and for waiting and signaling (notification) threads[14] is also delegated to the programmers hands. However, unlike the signaling mechanism of monitors and POSIX-THREADS, JAVA in its base mechanism lacks the possibility for fine-tuning the signaling sending process taking into consideration different synchronization conditions. There is a single waiting and signal mechanism which applies to the whole object (using POSIX-THREADS terminology, there is only one condition variable per object). Hence, a notify signal awakes a waiting thread regardless of its waiting condition (increasing the probability of spurious thread awaking). If there are multiple threads awaiting for different synchronization conditions, there is the possibility for a signal to awake the wrong processor, hence missing a correct destination. Due to this problem in JAVA, for safety concerns, it is advised to use broadcasts (signals to all waiting processors) instead of signaling a unique processor. This strategy leads to more possible spurious thread awaking, increasing the inefficiency of this mechanism.

In this article we are interested in implementations of automatic statically safe conditional synchronization mechanisms. Hoare [16] presents a possible automatic awakening mechanism (based on conditional waits), which he acknowledges as being simpler than condition variables, but with much less efficient implementations. That algorithm consists on sending broadcast signals after the execution of any of the external concurrent object's services, in order to impose a verification of all conditional waits in all waiting processors.

[14] Since we are now referring to JAVA, to avoid misunderstandings we will use its thread terminology instead of processor.

This algorithm can be made much more efficient, if the programming language has the ability to distinguish commands and queries. In that case, it is necessary to signal all waiting processors, only after the execution of commands (and eventually, also after non-pure queries), as they are the only ones who may change the object's state.

Currently this is the approach that we have been following, in the implementation of MP-EIFFEL.

In this specific problem, the operational approach to concurrency – in which programmers are required to program directly the synchronization of concurrent objects – despite its unsafety, is still the one which allows much more efficient implementations of conditional synchronization awaking algorithms.

Nevertheless, two possible alternative approaches are being studied which may provide safe algorithms to this problem, which approximate the efficiency of hand made programmer algorithms.

One approach, extends the usefulness of concurrent assertions also for signaling purposes. If a concurrent assertion in a precondition is required to be a conditional waiting mechanism, then their occurrence in postconditions or in other internal assertions (such as checks), could be used for the complementary awaking process. In the example of concurrent stacks given above, a `pop` service is required to conditionally wait for a non-empty stack. However, the postcondition of the `push` service is precisely the non-emptiness of the stack, hence it seems a perfect fit. A possible algorithm to implement this approach, consists in assigning condition variables to each different concurrent assertion in preconditions. Then, the waiting process in those preconditions would simply be a wait operation on the respective condition variable. The signaling process, on the other hand, requires a little more work. First it is necessary to express all possible concurrent assertions (in preconditions, postconditions or in other assertions[15]) in relation to their individual concurrent assertion clauses. The signaling algorithm could then take advantage of that static knowledge, to signal only the conditional variables in which its precondition contains identical assertion clauses as those in the signaling postcondition or check assertion.

This approach, though promising, has the drawback of requiring appropriate use of concurrent assertions by the programmer, which may pose problems if a normal class (such as a stack) is to be reused for concurrent objects.

Another promising approach, requires a deeper introspection inside the object's code. The idea is to attach to each of the object's services, the static set of attributes which may be modified by the service execution. Likewise, a set of depending attributes is attached to each concurrent assertions of preconditions. The algorithm would then be to signal, at the end of each service, all the preconditions which may depend on the attributes possibly modified by the service.

[15] Except invariants, as they are required to be always observed in the object's stable times.

Acknowledgments

I would like to thank João Rodrigues and Tomás Oliveira e Silva for their invaluable help and suggestions. I would also like to thank all the reviewers for their comments and suggestions which have improved the clarity and contents of the article.

References

1. Oliveira e Silva, M.: Concurrent object-oriented programming: The MP-Eiffel approach. Journal of Object Technology: Special issue: TOOLS USA 2003 **3**(4) (2004) 97–124
2. Meyer, B.: Eiffel: The Language. Prentice Hall, Englewood Cliffs, N.J. (1992) 2nd printing.
3. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice Hall (1997)
4. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Third edn. Addison-Wesley (2005)
5. Stroustrup, B.: The C++ Programming Language. third edn. Addison Wesley Longman (1997)
6. Agha, G.A.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, Massachusetts (1986)
7. Meyer, B.: Systematic concurrent object-oriented programming. Communications of the ACM **36**(9) (1993) 56–80
8. U.S. Government: Ada 95 Reference Manual (Language and Standard Libraries). (1995)
9. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley (1997)
10. Kramer, R.: iContract-the JavaTM design by ContractTM tool. In: Proceedings of Technology of Object-Oriented Languages – TOOLS 26. (1998) 295–307
11. Edwards, S., Sitaraman, M., Weide, B., Hollingsworth, E.: Contract-checking wrappers for C++ classes. IEEE Transactions on Software Engineering **vol.30**(11) (2004) 794–810
12. Herlihy, M.P., Wing, J.M.: Axioms for concurrent objects. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press (1987) 13–26
13. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492
14. Oliveira e Silva, M.: Concurrent contracts and inter-object synchronization in MP-Eiffel. Draft version available at `http://www.ieeta.pt/~mos/pubs` (2006)
15. Brinch Hansen, P.: Java's insecure parallelism. ACM SIGPLAN Notices **34**(4) (1999) 38–45
16. Hoare, C.A.R.: Monitors: an operating system structuring concept. Communications of the ACM **17**(10) (1974) 549–557
17. Brinch Hansen, P.: Monitors and concurrent pascal: a personal history. In: The second ACM SIGPLAN conference on History of programming languages, ACM Press (1993) 1–35
18. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. First edn. Addison-Wesley (1996)
19. Lea, D.: Concurrent Programming in Java. Second edn. Addison-Wesley (2000)

20. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with "readers" and "writers". Communications of the ACM **14**(10) (1971) 667–668
21. Lamport, L.: Concurrent reading and writing. Communications of the ACM **20**(11) (1977) 806–811
22. Peterson, G.L.: Concurrent reading while writing. ACM Trans. Program. Lang. Syst. **5**(1) (1983) 46–55
23. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) **13**(1) (1991) 124–149
24. Anderson, J.H., Jain, R., Ramamurthy, S.: Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97). (1997) 111–122
25. Sun Microsystems, Java Specification Requests: JSR166: Concurrency Utilities (2004) (`http://www.jcp.org/en/jsr/detail?id=166`).
26. Herlihy, M.: A methodology for implementing highly concurrent data structures. In: PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming, ACM Press (1990) 197–206
27. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Transactions on Programming Languages and Systems (TOPLAS) **15**(5) (1993) 745–770
28. Herlihy, M., Luchangco, V., Moir, M., William N. Scherer, I.: Software transactional memory for dynamic-sized data structures. In: PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing, ACM Press (2003) 92–101
29. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, ACM Press (2003) 388–402
30. Joung, Y.J.: Asynchronous group mutual exclusion. Distributed Computing **13**(4) (2000) 189–206
31. Oliveira e Silva, M.: Thread-Safe SmallEiffel (version beta-4) (2003) (`http://www.ieeta.pt/~mos/thread-safe-se/index.html`).

A Example Code

The code presented here is pure testable EIFFEL and was compiled with the thread-safe SmallEiffel package developed by the author (available in [31]).

A.1 Stack

```
-- Generic unbounded STACK class

deferred class STACK[E]

feature

  count: INTEGER is
      -- Number of elements
    deferred
    end;

  empty: BOOLEAN is
    do
      Result := count = 0
    end;

  top: E is
      -- STACK's last pushed element
    require
      not empty
    deferred
    ensure
      same_count:   count = old count
    end;
```

```
  push(elem: like top) is
    deferred
    ensure
      one_more:   count = old count + 1;
      element_placed_in_the_top:   top = elem;
    end;

  pop is
    require
      not empty
    deferred
    ensure
      one_less:   count = old count - 1
    end;

invariant

  count >= 0;
  empty = (count = 0)

end -- STACK
```

A.2 Stack: Monitor

```
class MONITOR_STACK[E]

creation
  make

feature {NONE}

  stack: STACK[E];
  mtx: MUTEX;
  cnd_var: CONDITION_VARIABLE;

feature

  make(s: STACK[E]) is
    require
      s /= Void
    do
      stack := s;
      create mtx.make;
      create cnd_var.make
    end;

feature

  count: INTEGER is
    do
      mtx.lock;
      Result := stack.count;
      mtx.unlock
    end;

  empty: BOOLEAN is
    do
      mtx.lock;
      Result := stack.empty;
      mtx.unlock
    end;

  top: E is
    do
      mtx.lock;
      from until not empty loop
        cnd_var.wait(mtx)
      end;
      Result := stack.top;
      mtx.unlock
    end;

  push(elem: like top) is
    do
      mtx.lock;
      stack.push(elem);
      mtx.unlock;
      cnd_var.broadcast
    end;

  pop is
    do
      mtx.lock;
      from until not empty loop
        cnd_var.wait(mtx)
      end;
      stack.pop;
      mtx.unlock;
      cnd_var.broadcast
    end;

end -- MONITOR_STACK
```

A.3 Stack: Readers-Writer Exclusion

```
class RW_EXCLUSION_STACK[E]

creation
  make

feature {NONE}

  stack: STACK[E];
  rwl: READ_WRITE_LOCK;
  mtx: MUTEX;
  cnd_var: CONDITION_VARIABLE;

feature

  make(s: STACK[E]) is
    require
```

```
      s /= Void
    do
      stack := s;
      create rwl.make;
      create mtx.make;
      create cnd_var.make
    end;

feature

  count: INTEGER is
    do
      rwl.read_lock;
      Result := stack.count;
      rwl.read_unlock
    end;

  empty: BOOLEAN is
    do
      rwl.read_lock;
      Result := stack.empty;
      rwl.read_unlock
    end;

  top: E is
    do
      rwl.read_lock;
      from until not empty loop
        rwl.read_unlock;
        mtx.lock;
        cnd_var.wait(mtx)
        mtx.unlock;
        rwl.read_lock;
      end;
      Result := stack.top;
      rwl.read_unlock
    end;

  push(elem: like top) is
    do
      rwl.write_lock;
      stack.push(elem);
      rwl.write_unlock;
      cnd_var.broadcast
    end;

  pop is
    do
      rwl.write_lock;
      from until not empty loop
        rwl.write_unlock;
        mtx.lock;
        cnd_var.wait(mtx)
        mtx.unlock;
        rwl.write_lock;
      end;
      stack.pop;
      rwl.write_unlock;
      cnd_var.broadcast
    end;

end -- RW_EXCLUSION_STACK
```

A.4 Stack: Readers-Writer Concurrent (Lamport)

```
class RW_CONCURRENT_LAMPORT_STACK[E]

creation
  make

feature {NONE}

  stack: STACK[E];
  mtx: MUTEX;
  writer_in,writer_out: INTEGER;
  cnd_var: CONDITION_VARIABLE;

feature

  make(s: STACK[E]) is
    require
      s /= Void
    do
      stack := s;
      create mtx.make;
      create cnd_var.make
    end;

feature

  count: INTEGER is
    local
      success: BOOLEAN;
      v: INTEGER
    do
      from until success loop
        v := writer_in;
        Result := stack.count;
        success := v = writer_out
      end;
    rescue
      if v /= writer_out then
        retry
      end
    end;

  empty: BOOLEAN is
    local
      success: BOOLEAN;
      v: INTEGER
    do
      from until success loop
        v := writer_in;
        Result := stack.empty;
        success := v = writer_out
      end;
    rescue
      if v /= writer_out then
        retry
      end
    end;

  top: E is
```

```
    local
      success: BOOLEAN;
      v: INTEGER
    do
      from until success loop
        v := writer_in;
        from until not empty loop
          mtx.lock;
          cnd_var.wait(mtx)
          mtx.unlock;
        end;
        Result := stack.top;
        success := v = writer_out;
      end;
    rescue
      if v /= writer_out then
        retry
      end
    end;

  push(elem: like top) is
    do
      mtx.lock;
      writer_in := writer_in + 1;
      stack.push(elem);
      writer_out := writer_out + 1;
      mtx.unlock;
      cnd_var.broadcast
    end;

  pop is
    do
      mtx.lock;
      from until not empty loop
        cnd_var.wait(mtx)
      end;
      writer_in := writer_in + 1;
      stack.pop;
      writer_out := writer_out + 1;
      mtx.unlock;
      cnd_var.broadcast
    end;

end -- RW_CONCURRENT_LAMPORT_STACK
```

B Mixed Concurrent Schemes

B.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_SYNCH;
#define INVARIANT_SYNCH_INIT \
  {0,0,0,0,PTHREAD_MUTEX_INITIALIZER,PTHREAD_COND_INITIALIZER}

int command_test_invariant(int (*inv)(void *obj),void *obj,
                           INVARIANT_SYNCH *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_SYNCH *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;
}
```

B.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(...);
```

B.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(...);
```