What Would I Do For a SCOOP?

Arnaud Bailly

École Nationale Supérieure des Télécommunications, Paris, France
arnaud.bailly@enst.fr

Abstract. This is a white paper on what I think should be the future of SCOOP [Mey93,NAM03]. It contains two sections: first a summary of the research results I would expect to attain, and second a gathering of thoughts and remarks about the current state of affairs and the future of SCOOP.

1 In a Nutshell

1.1 Which way to go?

Starting from the current informal description and existing implementation for SCOOP, I suggest that essentially four topics should be explored:

- sharpening the definitions of SCOOP's philosophy and programming mechanisms for distributed computing, specifically by re-visiting the role of pre-conditions, post-conditions, and invariants (see Section 2.1);
- provide a (operational and/or denotational) formal semantics for a reasonable subset of Eiffel/SCOOP constructions, allowing comparison with existing models for concurrent objects;
- devising a verification method and implementing an automatic or semi-automatic prototype verifier convincing enough to integrate the EiffelStudio ISE (see Section 2.3);
- completing the SCOOP implementation to reach practicality so that reasonably-sized experiments can be carried out, yielding more insight about the model as well as benchmarking information.

Of course, the studies around those topics shall have strongly intertwined fallouts; in particular the refinement of the original SCOOP model shall greatly influence the implementation effort. In the remaining of this section, I review the key issues concerning the above topics in (very little) more details.

1.2 Model-Related Issues

It should first be stated that a clear revision of the SCOOP model is necessary. It indeed appears that, while going from a sequential to a concurrent model, the role of pre-conditions has changed dramatically. This is certainly imputable to the fact that making the sequential/concurrent transition implies switching to an interaction-centered activity model that replaces the computation-centered (value-returning, that is) sequential activity model [WG02,MP91]. The role of post-conditions is in particular very unclear in the concurrent model: post-conditions are supposed to ensure that some (safety) property is verified upon termination of a routine execution, but this information is of little relevance in a concurrent execution. Indeed, the fact

that concurrent objects *react* to the stimuli they receive during an unlimited amount of time brings *liveness* properties into the picture; among liveness properties are the absence of deadlocks among objects, the progress of the overall system, and the absence of starvation. This has for result that pre-conditions and post-conditions seem to belong to different worlds when concurrency is involved, the leading role being hold by pre-conditions. A nice model should provide a more symmetric, elegant view of pre/post conditions.

The above considerations should lead to a formal-enough model of concurrent computations, that should be representable using an ad-hoc theory. It is too soon to commit to a given specification style (*e.g.* process algebraic, set-theoretic, etc.). Looking in the literature, I could find no formalism close enough to support direct encoding of SCOOP in a natural, simple way; my intuition is however that the customization of an existing formalism to suit SCOOP's needs should be a valuable path to follow. The semantical framework to be used will depend on the relation to be established with existing formalisms (most have operational semantics, but some have denotational semantics). This semantic framework choice should furthermore be made while considering the analysis technique(s) that will be employed. Those techniques have strong constraints leaning on them: they should be largely compositional (because of dynamic binding of variables and polymorphism), and they should be simple enough to use by the practitioner. In consequence, they are quite likely to be only semi-automatic, the end user having to input information in order to complete the verification process. Furthermore, since many (liveness) properties shall be too hard to prove in the general case or even undecidable, it seems important to equip the implementation with appropriate mechanisms to cope with the weaknesses of static verification.

1.3 Implementation-Related Issues

Software developments shall be of prime importance to make SCOOP usable and to validate/revise the model of the previous section. In particular, I think that some mechanisms should be mandatorily available in SCOOP in order to reach maturity:

- extending the exception mechanism to cope with distribution, and providing a set of well-chosen exceptions related to distributed issues;
- implement algorithms detecting deadlock and the absence of progress;
- implement an efficient intra-object concurrency model;
- allowing distributed garbage collection to take place.

The real-time issues, not to be forgotten, should largely be tackled at implementation level. It shall be the goal of the revised SCOOP structure to allow an elegant integration of real-time issues into the programming model, and thus into the supporting software. Some candidate solutions shall be exposed in Sections 2.1 and 2.2.

1.4 Expected Results

The anticipated works should lead to several publications in the first year. I should essentially be in charge of theoretical developments concerning the formal theory and its semantics, that should constitute one publication. Another theoretical paper

could be devised about the way of proving classes in a distributed environment. I also intend to contribute to, at least, two publications about:

- the revision of the SCOOP programming model,
- the introduction of intra-object concurrency,
- the class-correctness verification tool,
- the implementation of run-time mechanisms as distributed exceptions and distributed deadlock detection, and experiments about them.

The real-time aspects could be integrated along several of those papers, potentially complemented by a full-fledge article dedicated to this topic.

2 Free-Jazz Thoughts

In this section, I put thoughts that may seem as disorganized and wild as a chorus by Sun Ra or Ornette Coleman. Hopefully, as in many free-jazz records, one should be able to find the underlying structure and motivation of the product, if I wrote it right enough. In the meantime, should one be offended by the dissonances, I deeply apologize.

2.1 Cleaning The Model

The will to "clean" the model should not imply that the current version is "dirty". There is however a certain interplay taking place between the *synchronization through pre-condition* and *object locking* mechanisms, interplay that should be stifled. This should be done by reducing the object locking mechanism to an implementation-directed way of ensuring object consistency, and not to a programming-directed tool to synchronize processes. But first, the main goal of this section is to convince the reader that the deep nature of assertions in sequential computations, as a byproduct of the "Design by Contract" methodology, has been lost in the concurrent interpretation. It is however explained how contracts can be tentatively recovered in concurrent environments.

Pre-Conditions Are Not Contractual Under Concurrency. Assertions (we shall consider only pre/post conditions) state under which conditions some data element (object) may be manipulated, and what is the result of the manipulation. This is sufficient for assertions to constitute contracts in sequential executions, an incorrect manipulation or implementation of an object having no possibility to eventually become correct. In concurrent executions, whenever a pre-condition is not respected by an invocation, there are two possible attitudes:

- as in the sequential case, immediately blame the object at the origin of the ivocation for not having agreed with other objects on a correct synchronization policy [Bai02], or
- allow delay incorrect manipulations until the target object reaches a state that is compatible with the request [RV00]. This may happen because some other concurrent object may interfere and change the state of the target object.

In SCOOP, the second solution has been retained. Hence, the pre-conditions are not contractual anymore, being used only for synchronization purposes. As a token of this non-contractual nature, no exception for pre-condition violation may be raised during a separate invocation.

Actually, the obligations induced by pre-conditions have been shifted from the theoretical level (bearing on object) to the meta-theoretical level (bearing on the semantic rules of the language). In practice, the obligation is on the compiler to generate code ensuring that the routine of a separate object is executed only when its pre-condition is verified.

Post-conditions have not changed in nature, and may still yield exceptions.

Distributed Object Rights and Obligations. The question is now: can pre-conditions recover their contractual nature, that was so elegantly worn? At this point, my answer would be: "yeah, sort of." Intuitively, the essence of the restoration of contracts would be to say *how long* an invocation agrees to wait before being treated by its target object. As object routines are guaranteed to be accessed properly by the satisfaction of their pre-condition, it means that the *safety* of routine invocations is assured, and only that the *liveness* aspects may be a motive of dissatisfaction for the invoking object. This dissatisfaction can in turn be exhibited by an object only if this object specified the wanted liveness properties bearing on the invocation before this invocation took place. Such a specification then forms a contract.

The liveness contracts of objects should be able to express the following requirements and obligations:

- an object may require to eventually get an answer to any routine invocation, even from a separate object;
- symmetrically, an object may require that an invocation of one of its routines be performed eventually;
- an object may require that some separate invocation of its routines will eventually be given a service turn;
- an object routine may bear an obligation to terminate (eventually).

It is obvious that such liveness specifications outcome the default expressive power of plain assertions. This is unfortunate, since it breaches the hope of being able to effortlessly transfer classes conceived for sequential computation into a distributed setting. However, it appears that the use SCOOP objects make of pre-conditions permit orthogonal specification of safety and liveness properties.

There is hence a need to find a suitable way of specifying liveness properties for objects. One may enhance pre/post conditions with specific notations (see Section 2.2), or use another interface language. Among likely candidates for this are variations on first-order logic (*e.g.* QL [BS97]), temporal logic (*e.g.* PLTL [MP91], TLA [Lam94]), Büchi fairness conditions, etc. Main considerations in choosing such a language should be modularity and re-usability of specifications, simple and multiple inheritance being of particular concern.

Locking and Synchronization. As mentioned earlier, there is an interplay between the object locking and the synchronization mechanisms. This interplay saliently reveals itself in [Mey97], page 1011, at the bottom. In the sentence:

> "*To exclude any client from accessing the resource, you must enclose the operations accessing the resource in a routine to which you pass it as an argument.*"

The goal of the example is to show how to program lockable resources through a semaphore-like mechanism. It is arguable that such synchronization should be done only through pre-conditions, and that the object locking mechanism, initially introduced to guarantee an always-consistent view of an object's attributed to the routines of that object, should be relinquished.

To obtain an exclusive access to an object, the pre-condition of a routine could be allowed to check for the sender of an invocation: a lockable resource would then put the reference to the owner in an `owner` attribute, and change its pre-condition to something resembling "`owner=invoker and ...`". Without the possibility of testing for the invoker of a method, a far-less elegant solution would be to add an argument to all the routines of a lockable object for the invokers to identify themselves.

Therefore, systematic object locking by invokers should be abandoned, because it is a generally too strong (and redundant) way of synchronizing objects; it furthermore disallows any intra-object concurrency, largely hampering execution efficiency. The suggestion of Piotr Nienaltowski [Nie03] to allow concurrent execution of reentrant (a.k.a. *pure*) routines is of course safe. But it implies letting exclusive locking obligation for non-pure routines, which is again a redundant way of synchronizing objects. Actually,

> an object should never take for granted that it has exclusive access to a resource by other means than an ad-hoc pre-condition mechanism.

Now, how can we allow the greatest amount of concurrency into an object's execution while keeping the semantics of assertions, that allowed the nice Design by Contract method for sequential objects? There is a formally-defined candidate that can be found in the literature: *linearizability* [AW94]. The definition of linearizability is given on a history-based model that relates concurrent accesses to a data element, each access having a duration (the time of its termination is posterior to the time of its start). A concurrent execution is linearizable if the data accesses can be reordered so that they are executed in a sequential order and they still yield the same results. This forms a natural way to extend the interpretation of abstract data types equipped with pre/post conditions to a distributed setting: by restricting to serializable executions, one may prove properties of the data type by using the usual proof systems devised for sequential executions. In [HW90], a (non-compositional) method is given to check that an implementation preserves the linearizability of a specification.

Linearizability can therefore be used in the following way:

- build a concurrent routine scheduler that ensures linearizability of executions;
- prove that any implementation (routine body) preserves linearizability.

It is not established whether linearizability is the most permissive (*i.e.* less-synchronizing) consistency criterion that features such properties. It is however know that sequentiality [AW94] is too permissive.

Should the implementation of a linearizability-enforcing scheduler be too costly, one may rely instead on standard proof systems for concurrency (such as [OG76,LS84]),

that allows to prove the valid execution of concurrent routines by showing that they interfere (*i.e.* modify variables) only in compatible ways. This would allow the user, after providing a suitable proof, to tag methods that may execute concurrently[1]. In a sense, this only widens the criterion exhibited in [Nie03].

Active *vs* Passive Objects. The distinction between active and passive objects is not explicit in SCOOP. However, active objects do exist: they are the ones that execute a method that will not terminate. The passive objects are resources that are accessed by other concurrent (active or passive) objects, and that have only routines that terminate. Passive objects present into a thread of execution are managed by a system scheduler, that determines what incoming separate invocations are to be served next. This yields two drawbacks. First, an active object (or the passive objects in the same thread) may not serve any separate invocation, as possible for example in Eiffel// [ACE95]. Furthermore, the termination problem being undecidable, there is no way of telling which objects are active and which are passive.

My intuition is however that making an explicit distinction between active and passive objects increases the inherent complexity of the model, without bringing any benefit. Classes of (really) passive objects should indeed have a certain, simple structure, and be conceived as passive from the beginning. The termination of their routines may often be proved by using loop variants (a.k.a *measures*). The fact that active objects may treat no invocation enforces their "active" nature, and it seems reasonable to think that only concurrently-shared data structures should be able to be manipulated by invocations, those structures being the mean used by active objects to synchronize or start remote computations, without need for them to invoke each other's routines.

As a final remark on that topic, one may notice that post-conditions are unable to specify the behavior of an active object, since the main routine of the object should terminate for the post-condition to be checked. This means again that we may need a stronger way of reasoning about concurrent classes.

2.2 Implementation Features and the Introduction of Real-Time

Introducing Real-Time. To introduce real-time aspects is a delicate task. There are two possible well-beaten paths that can be followed:

- to restrict language features to enhance predictability (as in, *e.g.* , [Sch00]) and perform heavy benchmarking, or
- to enhance or extend language features to improve reconfigurability and seamless degradation [DHS98,DFH+98,Cou99,CBCP01].

The first solution may support the decision to restrict the use of many object-oriented features, that introduce unpredictable delays:

- the routine lookup that comes with dynamic binding and polymorphism,
- virtual memory that allows abstract from the position of objects in the memory but introduces unpredictable page faults, and

[1] actually, we should use *compositional* proof systems, such as [Jon83,Stø91,PJ91]

– garbage collection.

I do not know if such a solution could offer an acceptable compromise between efficiency and object-oriented features. I would need a more in-depth study of the literature.

The other way is to enhance the reconfiguration capabilities of objects, so that they can adapt to different environments. The way is then to provide a time-bound specification with the code, so that the execution can be counter-checked against the specification (this has lately been tagged as model-carrying code [SRRS01]). In the case of real-time SCOOP objects, the specification should be able to express the four points identified in the "Distributed Object Rights and Obligations" section, page 4. Candidates for specification languages are again many, such as the extension of first-order logic with time of [Hoo94], the real-time temporal logic MITL [RT91], real-time process algebras such as π^δ [Bai02]. Another possibility would be to introduce timing aspects directly into pre and post-conditions (such as one or more clock variables, with reseting and value-testing capabilities), that would yield a model similar to timed automata [AD94]. In any way, the choice should again be made on pragmatic grounds, by considering reuse through inheritance as the paragonic goal.

Implementation Issues. After those considerations on real-time, I might add some others that are rather related to ground implementation problems. The first remark is that a notion of time may have to be taken into account sooner than later to make SCOOP tractable. Indeed, the four items of the "Distributed Object Rights and Obligations" section make use of the *eventuality* concept. This concept is at the heart of the definition of liveness properties, that have been so controversial in the early 1980's. The controversy settled on the fact that the respect or disrespect of eventuality properties can not be observed in finite time. This is because at any time, if an event that must eventually happen has not happened yet, then it can still happen later. Hence, if some object specifies that, when performing a separate invocation, this invocation must be served eventually, then the target object will never be accused of failing to its task because, could it speak, it could always argue: "I'll do it later!". Therefore, the progress of the system may not actually be enforced if we could only specify such eventuality properties. We have to come with a stronger cure.

This cure is of course to let objects specify that some invocation should be serviced before a given, fixed time. Such properties are called *bounded liveness* properties, and they validity can be determined in finite time. If such an invocation fails to be served on time, then the system may produce an exception, that the source object may catch. The other items of the object rights and obligations list can be treated in the same way as routine invocation. If this solution seems very natural, we should however not forget to explore its consequences. First, the method should be wrapped into a nice interface for it to be easily usable by the programmer. Indeed, it is the choice of timeout constraints, as for network protocols, that will determine the efficiency of the system. A default, usable policy should be proposed to the programmer that fits the most general needs. This policy should be tunable to cope with more specific needs. Second, using such a bounded-liveness specification sys-

tem enlightens the need for a clear definition and implementation of a distributed exception system in SCOOP.

On aspects not related to time, it seems important to integrate a distributed deadlock detection mechanism to SCOOP. In the same way that one may hope to statically solve all time-related correctness issues only in a few, well contained cases, we can not hope to avoid deadlocks and absence of progress in general (see Section 2.3). Deadlocks may occur because of two reasons:

- the implementation of a serializability-ensuring scheduler may not be wait-free;
- if a linearizable scheduler is not used, some routines may have to be executed with exclusive access their object's data, leading to potential cycles in the graph of locked resources (called the *wait-for* graph);
- synchronization through pre-conditions may lead to object states where no routine will ever be available for execution.

The last item above illustrates the natural elevation of the contractual status of post-conditions from ensuring safety properties to ensuring liveness properties. By comparing the post-condition of a currently-executing routine $\mathcal{R}$ to the pre-conditions of all the routines of the same class, we can infer the set of routines that will be available for execution after $\mathcal{R}$ has terminated. This can be used for instance to ensure the *fair* access of clients to an object, and to define *starvation-free* objects.

The detection of a deadlock, that should probably be carried on using a standard algorithm based on the wait-for graph data structure, should produce an exception in all the concerned objects, so that one or more of them may temporarily renounce to its hold on some data element and unlocks the system before renewing its request. The *duel* mechanism, as described in [Mey97], may be used as the basic mechanism for releasing locks.

Finally, though I already mentioned distributed garbage collection as a necessary passage to reach maturity, there are other points that are important and should be envisaged early in the conception for long-term developments, among them two may be cited: fault tolerance (at least support of unreliable networks), and object mobility/dynamic class discovery.

2.3 Proving Classes

As mentioned above, proving that classes keep their semantics when placed in a distributed environment can be done by using linearizability or by impeaching "bad" interferences to occur. This is true for classes of passive objects, where the termination then ensures the validity of the post-condition, but also for classes of active objects (such as the `Process` class of [Mey97]), where non-termination impeaches the post-condition to have any relevance.

Therefore, if classes of active objects are considered, to "prove" them will require proving liveness properties. A first duty should be to pick an adequate, reasonable definition of concurrent class correctness. There can be as many definition as liveness properties, depending in particular on the retained definition for the "progress" of a system. For example, there have been some works that allow to prove that invocations are eventually treated by their target object [RV00,Bou97,Yos02]. Each of those article define a notion of "unacceptable" message, a message that may not be treated

by the target object. All three definitions are different. What they have in common, though, is that the notion of error they use is in general quite weak (*i.e.* there may be "correct" systems that behave in an inappropriate fashion). Furthermore, they impose strong restrictions on name-transmission capabilities.

On the practical, verification side, the capacity that objects have to transmit names and create new objects implies that any theory allowing a faithful encoding of those two features will yield infinite-state models; many interesting properties shall therefore be undecidable. By using the provided formal language for SCOOP configurations, one may use model checking (for finite-state models only) or theorem proving, or a combination of both. A subset of infinite-state models could be tackled by performing safe but approximate abstractions on their state-spaces.

References

[ACE95] I. Attali, D. Caromel, and S. O. Ehmety. An operational semantics for the Eiffel// language. In *Actes des Journées du GDR Programmation, Grenoble*, 1995.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–236, April 1994.

[AW94] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994.

[Bai02] Arnaud Bailly. *Assume/Guarantee Contracts in a Timed Mobile Calculus.* PhD thesis, École Nationale Supérieure des Télécommunications, Dec 2002. available at `http://www.enst.fr/~abailly/`.

[Bou97] G. Boudol. Typing the use of resources in a concurrent calculus. *Lecture Notes in Computer Science*, 1345:239–??, 1997.

[BS97] G. Blair and J. B. Stefani. *Open Distributed Processing and Multimedia.* In Press. Addison-Wesley, 1997.

[CBCP01] Michael Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. *Lecture Notes in Computer Science*, 2218:160–??, 2001.

[Cou99] Geoff Coulson. A configurable multimedia middleware platform. *IEEE MultiMedia*, 6(1):62–76, – 1999.

[DFH+98] D. Donaldson, M. Faupel, R. Hayton, A. Herbert, N. Howarth, Kramer A., I. MacMillan, D. Otway, and S. Waterhouse. Dimma - a multi-media orb. In *Proc. Middleware '98*, The Low Wood Hotel, Ambleside, England, September 1998.

[DHS98] I. Demeure, F. Horn, and F. Singhoff. Automatic scheduling of a dynamic multimedia applications with polka: a case study. In *Fourth IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, pages 15–19, June 1998.

[Hoo94] Jozef Hooman. Extending hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–826, 1994.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LS84] Leslie Lamport and Fred B. Schneider. The Hoare Logic of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, April 1984.

[Mey93] Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, 1993.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Ed.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1997.

[MP91] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification.* Springer-Verlag, 1991.

[NAM03] P. Nienaltowski, V. Arslan, and B. Meyer. SCOOP: Concurrent programming made easy. in process of submission, 2003.

[Nie03] P. Nienaltowski. Extending the access control policy for scoop. in process of submission, 2003.

[OG76] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976. Papers from the Fifth ACM Symposium on Operating Systems Principles (Univ. Texas, Austin, Tex., 1975).

[PJ91] P. K. Pandya and M. Joseph. P-A logic - A compositional proof system for distributed programs. *Distributed Computing*, 5(1):37–54, 1991.

[RT91] R. Alur and T.A. Henzinger. Logics and Models of Real-Time: A Survey. In *Real Time: Theory in Practice*, volume 600 of *LNCS*, pages 74–106. Springer-Verlag, 1991.

[RV00] A. Ravara and V. Vasconcelos. Typing non-uniform concurrent objects. In C. Palamidessi, editor, *CONCUR'00*, volume 1877 of *Lecture Notes in Computer Science*, pages 474–488. Springer-Verlag, 2000.

[Sch00] D. C. Schmidt. Real time CORBA with TAO (the ACE ORB). Technical report, Washington University in Saint Louis, http://www.cs.wustl.edu/schmidt/TAO.html, 2000.

[SRRS01] Sekar, Ramakrishnan, Ramakrishnan, and Smolka. Model-carrying code (MCC): A new paradigm for mobile-code security. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2001.

[Stø91] Ketil Stølen. A method for the development of totally correct shared-state parallel programs. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 510–525, Amsterdam, The Netherlands, 26–29 August 1991. Springer-Verlag.

[WG02] Peter Wegner and Dina Goldin. Computation beyond turing machines. *Communications of the ACM*, 2002. To be published.

[Yos02] Nobuko Yoshida. Type-based liveness guarantee in the presence of nontermination and nondeterminism. MCS 2002-20, University of Leicester, April 2002.