Concurrent Behavior: A Construct to Specify the External Behavior of Objects in Object Databases

NICOLETTA DE FRANCESCO AND GIGLIOLA VAGLINI
Dipartimento di Ingegneria dell'Informazione, Università di Pisa, Via Diotisalvi, 2, 56126 Pisa, Italia

Abstract. We present a linguistic construct to define concurrency control for the objects of an object database. This construct, called concurrent behavior, allows to define a concurrency control specification for each object type in the database; in a sense, it can be seen as a type extension. The concurrent behavior is composed by two parts: the first one, called commutativity specification, is a set of conditional rules, by which the programmer specifies when two methods do not conflict each other. The second part, the constraint specification, is a set of guarded regular expressions, called constraints, by which the programmer defines the allowed sequences of method calls. At each time during an actual execution, a subset of constraints may be active so limiting the external behavior of the object. A constraint becomes active when its guard is verified, where a guard is composed of the occurrence of some method call $m$ along with the verification of a boolean expression on the object state and the actual parameters of $m$. A constraint dies when a string of the language corresponding to the regular expression has been recognized. While the commutativity specification is devoted to specify the way in which the external behavior of an object is influenced by the existence of concurrent transactions in the system, the constraint specification defines the behavior of the object, independently from the transactions. Since the two parts of the concurrent behavior are syntactically distinct and, moreover, each of them consists of a set of independent rules, modularity in specifying the objects is enhanced, with respect to a unique specification. We outline an implementation of the construct, which is based on a look-ahead policy: at each method execution, we foresee the admissible successive behaviors of the object, instead of checking the admission of each request at the time it is actually made.

Keywords: object oriented databases, concurrency control, concurrent transaction

1. Introduction and motivations

The introduction of object-orientation in database languages [3, 13, 24] has made suitable an approach in which each object has the responsibility of its own control. From the point of view of increasing concurrency in method executions, each object can be equipped with some specification of the commutativity between its methods. This commutativity in general depends on the semantics of the object itself. There are many proposals following this line [10, 11, 14, 15, 22, 26], where a notion of semantics based commutativity, and consequently a semantic notion of serializability, are defined. In [7], the same direction is followed with a linguistic point of view: in fact, a construct to specify commutativity is introduced and added to the type definition of an object inside the object-oriented database language Nuovo Galileo [1].
From the other hand, languages have been developed [18, 19, 25] to specify the correct sequences of method invocations for an object. These languages resemble those for specifying concurrent processes like as regular expressions, CCS [16] and CSP [11] and essentially describe the external behavior of the objects without taking into account serializability problems. For example, given a data type Queue, it is possible to specify that, when it is empty, a remove operation is not allowed, while it is possible to perform an insertion. This kind of specification does not concern transactions, but it expresses the consistency properties of the object.

We think that the two aspects of the specification of an object are equally important; for this purpose, we have extended the construct presented in [5] in order to face both aspects. The construct, called concurrent behavior, is composed of two parts. The first one is the commutativity specification, i.e., a set of independent conditional rules by which the programmer specifies when, from her or his observational point of view, two methods can be executed in any order without producing appreciable differences in the expected result or in the object final state. The construct permits to achieve a high degree of concurrency because of the expressive power in the specification of conflicts. In fact, in order to specify finer conditions on method commutativity, the state of an object can be inspected by means of the invocation of any of its methods. The approach of basing method commutativity on the object state is more general than that of using only the results of the invoked method: in fact, while this last approach can be easily simulated by the former one, the converse is not necessarily feasible.

The second part of the behavior specification, more specifically addressed in this paper, consists of a set of independent guarded regular expressions, each one specifying a constraint on the external behavior of the object. In general, a constraint may regard some substrings of the whole behavior of an object. At each time, during an actual execution, several constraints may be contemporarily active: a constraint becomes active when a method is executed and/or a certain condition occurs; the constraint dies at the occurrence of a specified method call. The entry conditions are specified by the programmer for each constraint; as in the case of commutativity, they may be very general. Thus all the sequences of allowed method calls are the strings of a language defined by both the conditional rules and the regular expressions.

The approach of independently specifying a set of independent rules concerning commutativity and behavioral constraints enhances modularity in specifying the object, with respect to a unique specification containing parallel and sequential operators (like as a path expression, or a CCS specification, for instance). In this way a programmer can modify the behavior of an object only by adding or removing a rule. The syntactic distinction between the commutativity and the constraint specification, besides inducing differently structured implementations, corresponds to a particular methodology: a programmer first specifies what methods are generally independent, and after specifies the particular contexts in which they can be dependent on each other. Furthermore, we can see the