Rule Execution in CPLEX: A Persistent Objectbase

Meichun Hsu                Thomas E. Cheatham Jr.

Aiken Computation Laboratory
Harvard University

1. Introduction

There has been much interest in recent years in extending the basic database systems with abilities to handle data constructs of richer semantics and more flexible structures, to embed more powerful data manipulation languages, and to execute rules, or triggers, in addition to "user transactions". CPLEX [Che88a, Che88b] is one such system currently under implementation at Harvard University. CPLEX is essentially a language for defining and manipulating a persistent, shared, object base. The data model and data language is based on the DAPLEX functional data model and language [Shi88]. In addition, CPLEX supports two classes of rules.

While the ability to execute (i.e., to fire) rules asynchronously is considered a desirable extension for database systems [Esw76, Sto86, Syb87 Coh86], not much work has been done in defining the precise semantics of concurrent execution in a system capable of firing rules asynchronously. It is suspected that such an execution model for rules may require an extension to the basic transaction model.

In this paper we will focus on the rule model in CPLEX. Rules in CPLEX are classified into consistency rules and automation rules. The execution semantics of these rules are precisely defined in the language. We will also show that the CPLEX rule execution model maps nicely to a subset of an execution model proposed recently [HLM88] in the context of the Hipac system, a High Performance Active Database under development at Computer Corporation of America, although these two models were designed independently. This gives some evidence to the generality of the model proposed in [HLM88]. However, we also observe the need for the Hipac model to be extended and refined. In general, these observations may serve as a reference point for future designs of database systems that are to have rule execution capabilities.

The structure of the paper is as follows. In Section 2 we briefly describe the CPLEX object model and language. In Section 3 we present the rule execution model in CPLEX. Section 4 provides a comparison between the CPLEX rule model and that of the Hipac system, and points out additional issues to be addressed for rule execution.

2. Overview of the CPLEX Objectbase and Atomic Transactions

CPLEX is a prototype persistent objectbase under implementation at Harvard University. In this section we will provide a brief introduction to its object constructs and its query language. The description is not intended to be a complete specification of CPLEX. A complete specification for its syntax (grammar rules), semantics, evaluation environment, and implementation strategy can be found in [Che88a].

2.1. Objects in CPLEX

CPLEX is an extension of the programming language C. Its object model is essentially based on the DAPLEX functional data model.

Primitive built-in objects in CPLEX include integers, strings, atoms (as in Lisp), and entities, where entity corresponds to the conceptual notion like "person" or "car". An additional type of built-in objects are lists whose leaves are integers, strings, atoms and entities. The type system in CPLEX is extensible.

The object base consists of relations, which are conceptually sets of tuples whose components are objects. Unary relations correspond to classes and a class may be a subclass of another class.

Example. The following are unary relation declarations:

DECLARE Person() => ENTITY
DECLARE Student() => Person

A relation, $R$, may be thought of as a set of tuples, $<d_{1},...,d_{n},r_{1}>,...,<d_{M},...,d_{M},r_{M}>$. This same relation may be thought of as a function mapping a set of domain points, $<d_{1},...,d_{n}>,...,<d_{M},...,d_{M}>$ to their respective range values. The mapping can be single-valued (=>) or multi-valued (=>>) and partial or total.
Example. The following are some relation declarations:

DECLARE CourseTaken(Student) => Course
DECLARE Name(Person) => STRING
DECLARE Grade(Student, CourseTaken(Student)) => INTEGER

The "functional form" lets us write constructs like:

FOR EACH Course(Student S.T. Name(Dept(Student)) = "Computer Science") ...

2.2. Update Transactions in CPLEX

An update conceptually specifies the addition or deletion of tuples for a relation. We first illustrate updates in CPLEX through an example.

Example. The following CPLEX code changes the Address of a department entity, adds a new course to the department, and removes another course from the department:

FOR THE department SUCH THAT Name(department) = "Computer Science"
BEGIN
    LET Address(department) <- "Aiken"
    INCLUDE Course(department) <- Course(S.T. Name(Course)="CS161")
    EXCLUDE Course(department) <- Course(S.T. Name(Course)="CS261")
END

An update is in general a list of update statements bracketed by BEGIN and END; update statements thus bracketed constitute an atomic update transaction. An update statement contains one of the three keywords LET, INCLUDE and EXCLUDE. Following the above example, let \( \text{d} \) be the department being updated. The LET statement removes all tuples of relation Address for the domain point \( \text{d} \) and add the tuple with range value "Aiken" for that domain point. Similarly, the INCLUDE statement adds tuples and EXCLUDE deletes tuples for the relation Course for the domain point \( \text{d} \). An update may have a controlling clause of the form FOR A NEW ENTITY. This construct provides the only way to introduce a new entity.

The interpretation of an update transaction is as follows:

All the individual additions and deletions of tuples are determined, providing an initial set of proposed modifications to one or more relations. If the proposed modifications are valid, then the affected relations are changed to reflect the modifications. Otherwise, the update is aborted with no changes being made.

We define what it means for a set of proposed modification to be valid next.

3. Consistency and Automation Rules in CPLEX

In general, there are a number of constraints that we wish to impose on the relations comprising some objectbase. A consistency rule in CPLEX consists of a trigger -- a predicate that may become TRUE as the result of an update -- and, when that predicate does become TRUE, it is assumed that a constraint that we wish to impose is violated. The trigger then fires, enabling an associated repair to take place. The purpose of the repair is to specify one or more updates whose intended effect is to make the relations satisfy the constraints. One of the motivations for capturing the constraints in rules is to localize the specification of such constraints in rules instead of embedding the constraint checking and repair actions in the logic of updates. Embedding these activities in the logic of updates implies that all updates must be revised whenever a new constraint is imposed.

We first illustrate with some examples. Suppose that we have the declaration:

DECLARE LoginName(Employee) => STRING

constraining LoginName to be, as a function, total and single-valued. Thus, adding a new Employee without also providing a login name for that Employee would result in an invalid atomic transaction. Suppose that we wished to provide, as a default value for the login name, the employee’s last name. What we can do is to define a consistency rule that "fires" whenever a new Employee was added without being given a login name so that we can add the default value. The following consistency rule does this:

PROHIBITED MissingLoginName
    ADD Employee S.T. NOT EXISTS LoginName(Employee)
REPAIR
    LET LoginName(Employee) <- LastName(Employee)