$\mathcal{O}$bject $\mathcal{L}$ogic $\mathcal{I}$ntegration: a Scheme for Multiparadigm Design and Programming

J.H.M. Lee
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
Email: jlee@cs.cuhk.hk

P.K.C. Pun
Department of Computer Science
University of Manchester
Manchester M13 9PL, United Kingdom
Email: ppun@cs.man.ac.uk

Abstract

In the past decade, there has been much research effort dedicated to combine the object-oriented programming paradigm and the logic programming paradigm. Most of this effort sheds light upon the philosophy of multiparadigm programming as a near ideal mental model for a wide class of problem domains. In this paper we propose a scheme for object and logic integration—the $\mathcal{OLI}$ scheme. This scheme contributes to the multiparadigm programming philosophy by putting forward a multiparadigm programming design methodology and describing a multiparadigm programming language. Above all, the $\mathcal{OLI}$ scheme integrates the object-oriented and the logic programming paradigms at the design and language levels with a precise and well-balanced interface so that each paradigm shares an equal and cooperating partnership in problem analysis and problem solving.

1 Introduction

In order to design and implement a complex software system, one needs to understand the system from two perspectives: the *analysis understanding* and the *synthesis understanding*. Analysis understanding concerns the understanding of individual components that make up a software system. Synthesis understanding concerns the understanding of the interrelationships among individual components that make up a software system. The object-oriented programming paradigm (Cox 1986; Booch 1986; Meyer 1988) tends to promote analysis understanding by decomposing a software system into components, each of which is reasonably sized and well focused. These coherent components are organized as class hierarchies, instances of which interact with each other by message passing. The logic programming paradigm (Kowalski 1979; Sterling and Shapiro 1986; Deville 1990) tends to promote synthesis understanding by making interrelationships among different components of a software system apparent. These relationships are expressed explicitly as logical statements from which other valid statements can be derived by logical deductions. In terms of software engineering, an integration of the object-oriented and the logic programming paradigms serves as a near ideal mental model which promotes both the analysis and synthesis understanding processes.

In this paper, we propose the $\mathcal{OLI}$ (for $\mathcal{O}$bject $\mathcal{L}$ogic $\mathcal{I}$ntegration) scheme for integrating the object-oriented and the logic programming paradigms with a precise and well-balanced interface. The term *interface* refers to a common boundary at which the object-oriented and the logic programming paradigms meet and communicate with each other. From the programming design point of view, this interface should be precise so that the analysis role played by the object-oriented paradigm and the synthesis role played the logic paradigm are clearly distinguished in the programming design process. In order to achieve this, we put forward a multiparadigm programming design methodology which emphasizes and separates the functional and relational behavior of objects in a problem domain. From the programming language point of view, this interface should be well-balanced so that constructs in one paradigm are not just emulated but coexist comfortably with constructs in the other paradigm. In order to achieve this, we describe a multiparadigm programming language in which objects are treated as an enrichment of the Herbrand universe (Lloyd 1987) while the set of predicate definitions is regarded as an object with deductive capabilities. Thus the intersection of the paradigms in the $\mathcal{OLI}$ language lies in the *enriched Herbrand universe* or the *object pool*, which serve as the semantic basis of the language. The major difference between the $\mathcal{OLI}$ scheme and other integration effort is that the $\mathcal{OLI}$ scheme is based on the concept of maintaining an equal partnership between the two paradigms rather than the concept of enhancing any one paradigm with constructs of the other. The $\mathcal{OLI}$ multiparadigm design methodology and the $\mathcal{OLI}$ multiparadigm programming language are complementary to each other in developing good multiparadigm programs which are natural to a wide class of problem domains. Another important consequence is that each paradigm in $\mathcal{OLI}$ performs tasks only of the paradigm's natural strength: object for modeling dynamic state changes and logic for modeling relationships in a static world.

The paper is organized as follows. Section 2 reviews related work. In section 3, we present a university course registration system as a motivating scenario for multiparadigm programming. Section 4 examines the interface between the object-oriented and the logic programming paradigms at the programming design level in the $\mathcal{OLI}$ scheme. A multiparadigm programming design methodology will be described in this section. In section 5, we define the interface between the object-oriented and the logic programming paradigms at the programming language level in the $\mathcal{OLI}$ scheme. A multiparadigm programming language based on an integration of SMALLTALK (Goldberg and Robson 1989) and PROLOG (Clocksin and Mellish 1984) will be described in this section. Section 6 concludes the paper with a summary and future research directions.

2 Related Work

In the past ten years, a large number of languages and systems have been proposed to combine the object-oriented and the logic programming paradigms. Although most of this effort sheds light upon the philosophy of multiparadigm programming as a near ideal mental model for a wide class of problem domains, the issue of an appropriate design methodology supporting multiparadigm programming has not been addressed thoroughly. In general, there are two kinds of strategy in integrating the object-oriented and the logic programming paradigms: the camp of logical emulation of objects and the camp of object-oriented emulation of predicates. In the first camp, objects are either emulated as unconditional clauses or perpetual processes. Languages and systems (Shapiro and Takeuchi 1983; Janson 1994; Zaniolo 1984; Gullichsen 1985) of this camp force a programmer to subscribe to the clausal view or the process view which do not necessarily agree with the analysis role played by the object-oriented programming paradigm in programming design. In addition, extra-logical constructs are employed to model state changes. More recently, a number of vanilla preprocessors, e.g. *L&O* (McCabe 1992) and PROLOG++ (Moss 1994) have been developed to reinstate the analysis role of the object-oriented programming paradigm. However, problems with the semantics of the translated programs still remain. Moreover, the motivation of *L&O* and PROLOG++ seems just to extend the logic programming paradigm with features from the object-oriented paradigm in or-

der to make the design and development of large logic programs manageable. Such a motivation does not advocate an equal and co-operating partnerships between the object-oriented and the logic programming paradigms which is regarded to be a more important objective for their integration.

In the second camp, class methods are allowed to be defined as a set of clauses. Languages and systems (Ishikawa and Tokoro 1986; Mellender 1988; Wu 1991; Pun and Kahn 1995) of this camp require logical predicates to be specified within classes, *i.e.* predicates do not exist independently. In general, languages and systems of these two camps are extensions of a native programming paradigm by emulating programming constructs of the other paradigm as an enhancement of the respective native paradigm. These multiparadigm languages and systems do not provide an equal and cooperating partnership between the object-oriented and the logic programming paradigms in both problem analysis and problem solving. Moreover, these emulation strategies always result in an interface with weak semantics. The interface is usually not well-balanced to support communication among constructs of these two paradigms.

The LIFE language (Aït-Kaci and Podelski 1993) subscribes to the logic programming, functional programming, and object-oriented programming paradigms. The language design of LIFE is most similar to the philosophy of our approach. The central component of LIFE is a calculus of ψ-terms (Aït-Kaci 1984; Aït-Kaci 1986), an extension of first order terms (Herbrand terms and λ-terms) which provides structural convenience for representing record-like data structures. The ψ-calculus defines a clear notion of subsumption interpretable as a subtype ordering, together with an efficient unification operation interpretable as type intersection. Thus ψ-terms account for a form of structured type inheritance typically found in object-oriented languages. The ψ-calculus, however, does not address the issue of dynamic state changes and it only accounts for a limited set of object-oriented features in terms of type inheritance.

3 A Multiparadigm Programming Scenario

In this section, we study the design and implementation issues of a university registration system. We examine the reasons why the object-oriented programming paradigm or the logic programming programming paradigm alone is not sufficient and effective to address these issues. It is hoped that this scenario study would provide enough evidence to motivate the need for an integration between the two paradigms.

3.1 A University Registration System

A university is an organization made up of different personnel. Each member of the university personnel has a name and other details such as age, sex, address, *etc*. The members of university personnel relevant in our scenario are lecturers, students and the registrar. Each lecturer is responsible for teaching certain university courses. Each student is identified by a student identification number; he/she is currently studying in a particular year and belonging to a particular major subject stream. The university registrar is responsible for managing information regarding course registration. The university offers a number of different courses in which students can choose to enroll according to the previous courses they have taken. Each university course is identified by a course code; it has such details as course contents and the relevant major subject stream it belongs to. A university registration system is an information system which maintains information of different courses offered by the university and manages the ways these courses are related to the university personnel. In order to prevent students from abusing this system, an access quota is imposed on students in order to monitor their usage of the system.

3.2 Object-Oriented Programming Design and Implementation

In the object-oriented programming paradigm, university personnel and courses are expressed as object classes in the university registration system. The members of the university personnel are further refined into lecturer, student and

registrar object classes, *i.e.* superclass/subclass relationships. Attributes belonging to objects of these classes are then identified. Methods belonging to these objects are decided based on the functionality of the university registration system. In general, these class classifications and representations are easily expressed with an object-oriented language or system.

It is mentioned above that the university registration system should manage information regarding the different ways university courses are related to the university personnel. This information is best perceived as relationships among objects of university course and university personnel classes. However, relationships among objects are not directly supported by object-oriented languages and systems. They are usually simulated by the use of instance variables and methods which refer to other objects, *i.e.* association and aggregation. In order to represent a particular relationship between student and course objects, *e.g.* information regarding courses taken by students in the past, we may have to associate an instance variable with each student object for this purpose. Two different methods may then be associated with the student objects for finding out all courses taken by them and for checking whether they have taken certain courses, although they are just two different applications of information inferred from the same relationship. Such an approach imposes an unidirectional application restriction to relationships which are multi-directional in nature. In some extreme cases, if a relationship must be traversed in more than one direction, an instance variable is required in each object participating in the relationship. Moreover, representing relationships among objects as instance variables and methods fails to capture the declarative semantics of these relationships. For example, the relationships between students and courses that they are eligible for is dependent on the relationship between students and courses taken by them in the past. If these two relationships are represented as instance variables and methods, their relational abstractions are lost and this makes the process of understanding and maintaining the relationships harder.

3.3 Logic Programming Design and Implementation

In the logic programming paradigm, the principle components of the university registration system are rules and facts about the university personnel and university courses. This paradigm subscribes to the *closed world assumption* that all that is true about the university registration system is what can be proved logically on the basis of these facts and rules. Facts are inferred from relationships which exist among the university personnel and the university courses, *e.g.* facts asserting a particular student has taken certain courses in the past. Rules represent derived relationships from existing relationships, *e.g.* a rule to deduce whether a student can take a particular course based on courses taken by him in the past. Logic programming languages support such an approach of relational abstraction. Since relational abstraction is declarative and multi-directional in nature, the same set of facts and rules can be reused in different contexts, *e.g.* the same piece of fact can be used to assert that a student has taken certain courses or to find out all courses taken by him in the past.

However, the only way for logic programming languages to describe a data type is to give an abstract, algebraic description of the properties and its operations as a set of facts and related rules. In this way, the structural aspects in the university registration system, *e.g.* name, age, access quota of a student, are represented in an abstract manner which makes them more difficult to understand than they should be. Moreover, update of these object structures, *e.g.* setting and resetting access quota of students, always involves using constructs of destructive effect or complicated semantics. In this way, it seems unnatural to represent history sensitive structures in the logic programming paradigm.

4 The $\mathcal{OLI}$ Multiparadigm Programming Design

In the previous section, we argue that each of the object-oriented and the logic programming paradigms alone is not sufficient and effective enough to address the design aspects of the university registration system. In this section, an

interface between these two paradigms at the design level is described to distinguish the analysis role and the synthesis role played by these two paradigms in the programming design process. A set of design guidelines are then formulated based on this interface. This set of guidelines represents a multiparadigm programming design methodology which integrates the object-oriented and the logic programming paradigms at the programming design level.

4.1 Multiparadigm Design Interface

In the $\mathcal{OLI}$ scheme, a system is composed of *objects* and *predicate definitions*, *i.e.*

$$\text{System} = \text{Objects} + \text{Predicates}.$$

Objects are tangible entities that exhibit some well-defined behavior. An object has state and functional behavior. Predicates in the system specify the relational behavior of all objects. The state of an object is a set of properties and their current values possessed by the object. Functional behavior of an object is a set of functions or procedures, *i.e.* methods, which effects *dynamic* state changes and message passing. Relational behavior of an object is a set of logical predicate definitions which specify valid relationships among objects in a *static* world. Other valid relationships can be derived by logical deductions. Figure 1 depicts the $\mathcal{OLI}$ multiparadigm design interface. In this interface, the object-oriented programming paradigm plays the analysis role by modeling the dynamic nature of state and functional behavior of objects in a system. The logic programming paradigm plays the synthesis role by modeling the static nature of state and relational behavior of objects in a system. The two paradigms meet at a boundary which constitutes the dynamic and static nature of the state of objects in a system. The separation of the dynamic nature and static nature of the state of objects is an important design principle of the $\mathcal{OLI}$ scheme. First of all, it provides a well-defined, equal and cooperating partnership between the object-oriented and the logic programming paradigms in the process of programming design. Moreover, the exclusion of dynamic state changes from the relational behavior of objects avoids many technical complications, *e.g.* keeping an infinite history of object state for backtracking, that may occur in a practical implementation.

The difference between functional and relational behavior of an object is a very subjective design exercise which relies much on intuition and heuristics. However, in most cases, such a distinction can be deduced from how the information inferred by a particular behavior is to be utilized. If we ignore the dynamic effect of the functional behavior to an object, it can be regarded as the relational behavior, the applications of which are specific and deterministic. Consider a set of predicates of the following form:

$$\mathcal{P}_{\mathcal{F}}(X_1, \ldots, X_i, \ldots, Y_1, \ldots, Y_j).$$

Arguments of the predicate $\mathcal{P}_{\mathcal{F}}$ are grouped in two sequences $X_1, \ldots, X_i$ and $Y_1, \ldots, Y_j$ such that for each sequence $X_1, \ldots, X_i$, there exists exactly one sequence of $Y_1, \ldots, Y_j$ so that $\mathcal{P}_{\mathcal{F}}$ is true. Applications of these predicates are usually specific and deterministic. The input and output status of arguments are well-defined. Thus predicates of the form of $\mathcal{P}_{\mathcal{F}}$ really are functions which map the arguments $X_1, \ldots, X_i$ onto the values $Y_1, \ldots, Y_j$. Because of their functional applications, predicates of the form $\mathcal{P}_{\mathcal{F}}$ are usually written as functions of the following form:

$$\mathcal{P}_{\mathcal{F}}(X_1, \ldots, X_i) = (Y_1, \ldots, Y_j)$$

and they are classified as contributing to the functional behavior of an object.

On the contrary, relational behavior is characterized by the set of predicates of the general form:

$$\mathcal{P}_{\mathcal{R}}(X_1, \ldots, X_n).$$

The sequence $X_1, \ldots, X_n$ are arguments of the predicate $\mathcal{P}_{\mathcal{R}}$ that makes $\mathcal{P}_{\mathcal{R}}$ true. Applications of these predicates are usually general and flexible. Any combination of the input and output status of arguments can be specified. On one hand, predicates of the form $\mathcal{P}_{\mathcal{R}}$ can be utilized to compute whether a known sequence of arguments forms valid relationships. On the other hand, these predicates can be utilized to compute the values of any subset of their arguments, given known values of the rest. Thus, taken all arguments altogether, the sequence forms valid relationships. In general, more than one solution set will be computed in the second case.

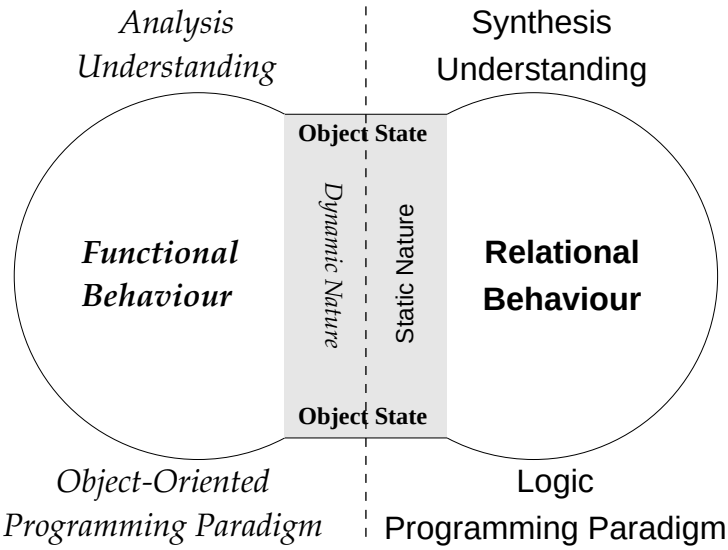

Figure 1: The $\mathcal{OLI}$ Multiparadigm Design Interface

Computation is non-deterministic. Predicates of this kind are classified as contributing to the relational behavior of an object. These predicates express relationships among objects explicitly while their applications are general and flexible.

4.2 Multiparadigm Design Methodology

This $\mathcal{OLI}$ multiparadigm design methodology tries to apprehend systems of any domain as composed of objects and predicate definitions. The fundamental principle of this design methodology is to enable the object-oriented and the logic programming paradigms to play their competent roles in the programming design process. The general design process of this design methodology can be characterized by the following steps:

1. Identify entities and their properties in a system.
2. Divide properties of each entity into its state, functional and relational behavior.
3. Organize states and functional behaviors of all entities in a system as different object class hierarchies.
4. Represent relational behaviors of all entities in a system as predicates involving objects.

Figure 2 shows the high-level design of our university registration system based on the $\mathcal{OLI}$ multiparadigm design methodology. Functional behavior is the only means to model state changes in objects. Other than that, the difference between the functional behavior and the relational behavior of an object is somewhat subjective. However, we can always make the distinction by the way in which a particular behavior is utilized in a system. Applications of functional behavior of entities are usually unidirectional. For example, behaviors of the student object which give its name, age, sex, address are classified as functional behavior of the student object. On the other hand, applications of relational behaviors of objects are usually multi-directional. For example, properties of the student entity such as courses taken by him in the past is specified using predicate `coursesTaken/2`. Representing relational behavior of an information system as predicates does not restrict the way in which these predicates are utilized. For example, the information about all courses taken by a particular student in the past and the information about whether a particular student has taken certain courses in the past can be inferred from the same

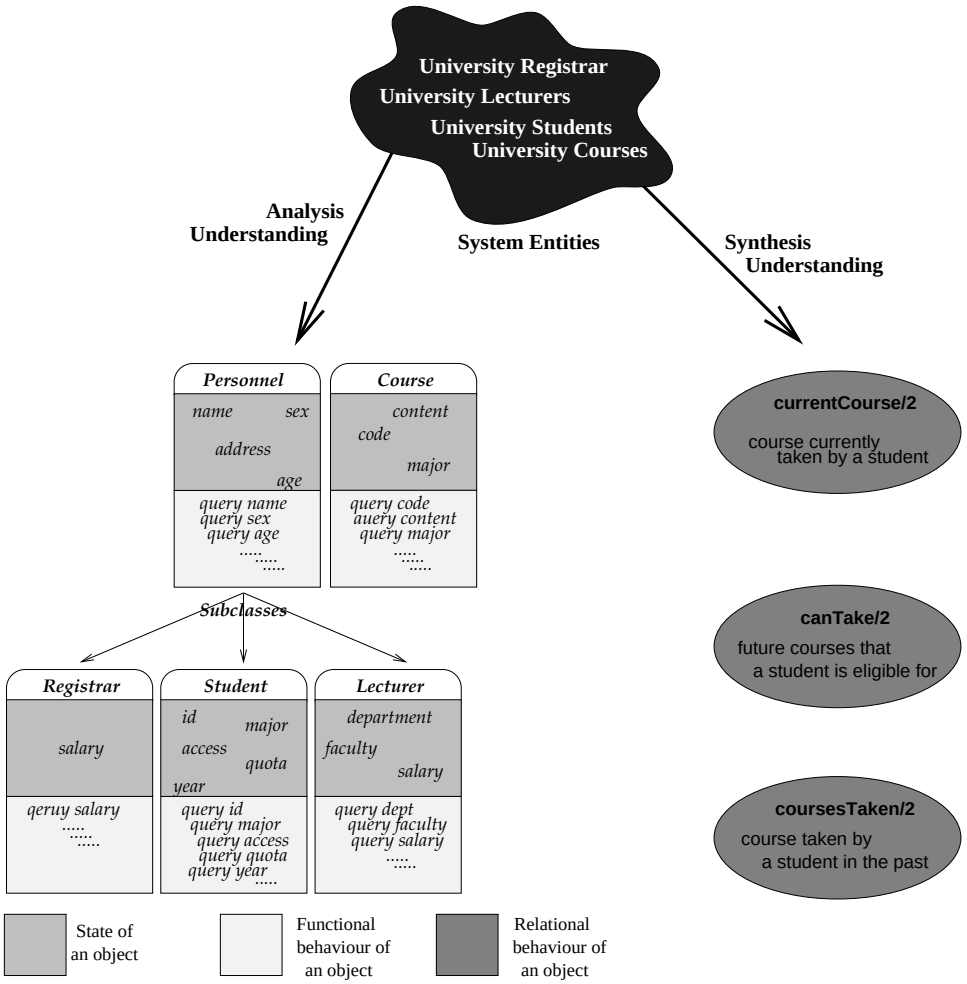

Figure 2: Decomposition of a University Registration System

`coursesTaken/2` predicate. Moreover, representing these relational characteristics explicitly as predicates among objects makes possible the declarative specification of derived predicates. New predicates can be defined based on the existing predicates. For example, the relationship between students and courses that they are eligible for can be defined as a predicate `canTake/2` using the existing predicates `coursesTaken/2`.

5 The $\mathcal{OLI}$ Multiparadigm Programming Language

In this section, an interface between the object-oriented and the logic programming paradigms at the language level is discussed. This interface is based on a collection of high-level programming constructs which can be interpreted meaningfully within the context of individual paradigms. A multiparadigm programming language $\mathcal{OLI}$ based on this interface is then described. The $\mathcal{OLI}$ language, in its current form, resembles a blend of PROLOG and SMALLTALK, which are representational languages from logic programming and object-oriented programming respectively. The framework of $\mathcal{OLI}$, however, by no means depends on any peculiar features of PROLOG and SMALLTALK. Our subsequent description remains valid if we replace Prolog and Smalltalk by any pure logic programming and object-oriented languages, such as Gödel (Hill and Lloyd 1994) and Eiffel (Meyer 1992). In the following, we give a brief overview of the $\mathcal{OLI}$ language. We conclude the section with an informal description of the $\mathcal{OLI}$ execution model, emphasizing on how control flow is passed between the object part and the logic part of an $\mathcal{OLI}$ program. At appropriate places, we give code examples to illustrate how $\mathcal{OLI}$ can be used to address the implementation aspects of the university registration system outlined in section 3 in a multiparadigm programming manner. We assume readers basic knowledge of PROLOG and SMALLTALK. Interested readers are referred to (Sterling and Shapiro 1986; Goldberg and Robson 1989) for detailed description of the languages.

5.1 Multiparadigm Language Interface

Logic programming languages concern the representation and deduction of relationships defined among individuals. These individuals are *terms*[1] in the Herbrand universe (Lloyd 1987) of a static world. Object-oriented programming languages effect state changes of individuals through message passing. These individuals are object instances of defined classes in a dynamic environment. An $\mathcal{OLI}$ program consists of two parts: the *logic* part and the *object* part. We argue that the precise interface between the object-oriented programming and logic programming paradigms at the language level lies at the collection of individuals as defined in the logic part and the object part of an $\mathcal{OLI}$ program. Figure 3 illustrates the interface as viewed from different perspectives. Different fonts are used to denote the different views of the interface from each of the two paradigms.

From the logic point of view, this collection of individuals constitutes an *enriched Herbrand universe* (**eHu**). The **eHu** is the result of augmenting the Herbrand universe, existed as a result of a logic program, with objects defined in the object part. To differentiate between the two kinds of individuals, we call the latter *o-terms* and the former *h-terms*. H-terms are ordinary terms in logic programming languages such as PROLOG. O-terms are semantic terms, the status of which are similar to numerals in PROLOG. However, state or attribute of an o-term is not explicitly recorded in the structure of the term. If we want to inquire the state of an o-term, we have to use messages instead of unification (Lloyd 1987). Since logic describes relationship in a static world, we do not allow state change of o-terms in the **eHu** during a logical derivation. Thus, only state-enquiring, but not state-changing, messages to o-terms are allowed during derivation. Such an interface agrees with the principle of the multiparadigm design methodology described in section 4 in that the dynamic and static nature of the state of an object are well distinguished and separated.

From the object point of view, the collection

[1] Terms in logic programming correspond to tree data structures in conventional programming languages.

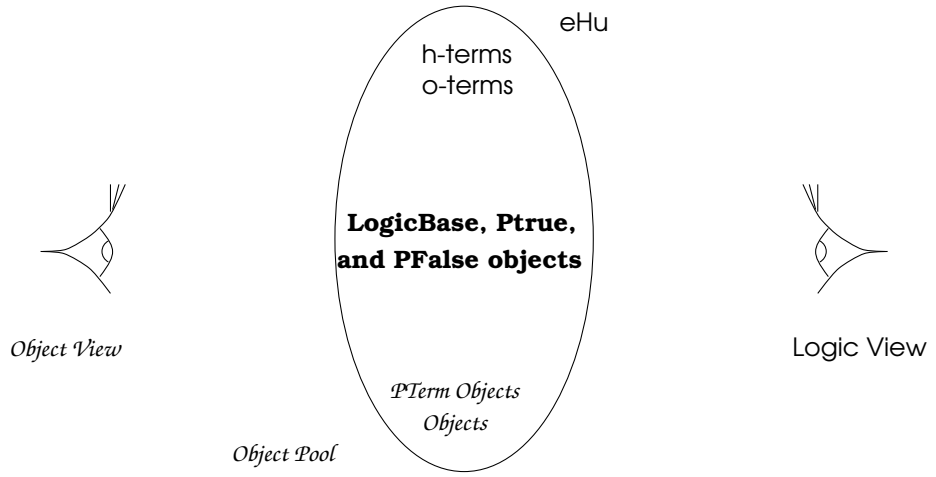

Figure 3: The $\mathcal{OLI}$ Programming Language Interface

of individuals at the interface is simply an *object pool* since Herbrand terms are also objects. In the $\mathcal{OLI}$ scheme, Herbrand terms are instances of the `PTerm` class, which provides methods to traverse as well as to update the structure of the terms. There are three special built-in classes in $\mathcal{OLI}$: `LogicBase`, `PTrue`, and `PFalse`. `LogicBase` has only one instance `logicBase`, which represents the logic part of an $\mathcal{OLI}$ program. The state of `logicBase` is the collection of clauses in the logic part. Thus the state of `logicBase` can be updated using the `assert:` and `retract:` messages. Another important message to `logicBase` is `query:`, which takes as argument a logical query and returns an instance of either the `PTrue` or `PFalse` class. `PTrue` is a subclass of the `True` class in SMALLTALK. Thus `PTrue` objects respond to the usual logical operations, such as `&` and `not`. `PTrue` objects has an additional state variable which contains the set of answer tuples of a successful logical derivation. The answer tuples can be retrieved using the `whileMoreDo:` message. The `PFalse` class is an alias of the `False` class of SMALLTALK, instances of which are returned by failed logical derivations. Instances of these special classes are part of the object pool (thus also the **eHu**). This design has interesting meta-programming implication, which is outside the scope of this paper.

5.2 The Object Part

Class definitions in $\mathcal{OLI}$ resembles that of Smalltalk, which consists of state variable declarations, and class and instance method definitions. We differentiate between *variant* and *invariant* methods. The former, besides computing a function, has the side-effect of creating new objects and/or changing states of some existing objects; while the latter is used for either state enquiry or function abstraction.

Figure 4 shows code fragments of the `Registrar` class definition. The `Registrar` class is a subclass of `Personnel`. `Registrar` has as instance variable `salary`, in addition to `name`, `age`, and `sex` inherited from `Personnel`. `Registrar` has one variant class method for creating new instances of the class and has no invariant class methods. A `Registrar` object has variant instance methods for adding and dropping courses for a student, advising a student on the kind of courses that (s)he can take, obtaining the class list of a course and the graduation list of a major stream. We examine the methods `add:course:` and `advise:` in more details.

The `add:course:` method takes as arguments a student and a course. The method starts by checking if the student has used up his/her CPU quota for accessing the registration system. If not, the method decrements the CPU quota of the student and inquires the logic part of the

```
Class Definition:
Personnel subclass: #Registrar
  InstanceVariableNames: 'salary'

Variant Class Methods:
name: aName age: anAge sex: aSex salary: aNumber
  "Create an instance of myself with age, sex, and salary."
  ...

Invariant Class Methods:

Variant Instance Methods:
...

add: aStud course: aCourse
  "Add a course for a student."
  (aStud cpuQuota > 0) ifTrue:
    [aStud deductCpuQuota.
     (logicBase query: [canTake(aStud,aCourse),curCourse(aStud,CList)]) ifTrue:
        [:cList|logicBase retract: [curCourse(aStud,cList)].
                logicBase assert: [curCourse(aStud,cList addHead: aCourse)]]].

advise: aStud
  "Advise a student on the kind of courses that (s)he can take."
  |list|
  list := PList new.
  (aStud cpuQuota > 0) ifTrue:
    [aStud deductCpuQuota.
     (logicBase query: [canTake(aStud,ACourse)]) whileMoreDo:
       [:aCourse|list addHead: aCourse].
     ^list]

...

Invariant Instance Methods:
salary
    "Answer my salary."
    ^salary
```

Figure 4: Code Fragments of the `Registrar` Class Definition in $\mathcal{OLI}$

university registration program to check if the student is eligible to take the course and the current courses that the student is taking. If the student can take the course under consideration, the method proceeds to update the current-course database encoded in the logic part of the program. A logical query is represented as an instance of the `Block` class. `CList` is a logical variable (using PROLOG's upper case letter convention) and denotes an unknown in the query. If the logical derivation activated by the query is successful, a `PTrue` object that contains a set of unary solution tuples for `CList` will be returned. The `PTrue` object returned can respond to the `ifTrue:` message with a block object with one argument. The argument of the block will take on the value of the first solution tuple, which is an instance of `PList`, a subclass of `PTerm`. The two statements inside the block serve to update the `curCourse/2` database in the logic part of the program. The `addHead:` message prepends a course object to the head of the `PList` object, held in variable `cList`. If the query fails, a `PFalse` object is returned.

The `advise:` method takes as argument a student and advises the student on the courses that (s)he can take. After the method has checked the student's CPU access quota, it enquires `logicBase` about the set of eligible courses for the student. If the set is non-empty, the unary solution tuples will be contained in the `PTrue` object returned from a successful logical derivation. The upper case `ACourse` is a logical variable. What is different in this case is that the `PTrue` object receives a `whileMoreDo:` message, which evaluates its argument block once for each solution tuple contained in the `PTrue` object. The argument block of `whileMoreDo:` collects the eligible course of the student in a `PList` object.

5.3 The eHu and the Logic Part

In the logic part, persistent objects can be defined and named using the classes defined in the object part of an $\mathcal{OLI}$ program. These persistent objects extend the Herbrand universe to form the **eHu**. They are the o-terms in the **eHu**. Example definitions of persistent objects in the university registration program are shown in figure 5. These definitions are made at the beginning of the logic part, followed by program code written in the style of Prolog rules.

Predicate definitions in the logic part consist of a set of *extended Horn clauses*, which are Horn clauses (Lloyd 1987) containing o-terms in addition to h-terms. The attribute of o-terms can be extracted by invoking invariant methods of the o-terms. In figure 6, we illustrate the syntax and informal semantics of extended Horn clauses using code fragments from the university registration program. The `coursesTaken/2`, `curCourse/2`, `preReq/2`, and `maxLoad/2` predicates define respectively the database of courses taken by individual students, courses currently enrolled in by individual students, the prerequisite courses of individual courses, and the maximum course load for students in each year of study. The `degreeReq/2` predicate defines the degree requirement of each major stream. The `canTake/2` predicate defines a binary relation on the eligible courses for individual students. It can be used to check whether a student can take a particular course and what course a student can take. A student can take a course if (1) (s)he has not reached the maximum course load, (2) (s)he has taken all the prerequisite of the course, and (3) the student's major stream matches the course's major stream. The definition of the `canTake/2` predicate reflects exactly the said rule. Note that the invariant message `year` and `major` are sent to the student object to inquire its year of study and major stream. Similarly, the invariant message `major` sent to a course object inquires the major stream of the course. The `length/2` predicate relates a list and its length. The `subset/2` is true if the first argument, represented as list, is a subset of the second. The `delete/3` predicate holds if the third argument list is obtained from removing the first argument from the second argument list. A student can graduate with a degree in a particular major if (1) the student's major stream matches with the input major stream and (2) the student has taken all the required courses of the major. Again, this rule is reflected clearly in the definition of the `graduationStudent/2` predicate.

5.4 The Execution Model

The $\mathcal{OLI}$ scheme advocates a truly equal partnership between the object-oriented and the

```
$tom =:= Student name:"Tom" age:20 sex:"M" year:1 major:"CSC" id:123 quota:10.
$mary =:= Student name:"Mary" age:22 sex:"F" year:2 major:"CSC" id:456 quota:10.
...
$jim =:= Lecturer name:"Jim" age:32 sex:"M" dept:"CSC" fac:"ENG" salary: 1000.
...
$csc4190 =:= Course code:"CSC4190" content:"AI" major:"CSC".
...
```

Figure 5: Code Fragments for Defining Persistent Objects in $\mathcal{OLI}$

logic programming paradigms. Programmers can compose a program entirely in the object-oriented style or in the logic programming style. In fact, SMALLTALK and PROLOG programs are special cases of $\mathcal{OLI}$ programs with only one part. Execution of $\mathcal{OLI}$ programs can be invoked either by sending a message to an object or by posing a logical query. $\mathcal{OLI}$ program execution is an interleaving of SMALLTALK style message sending and PROLOG style SLD-resolution (Lloyd 1987). In the following, we briefly explain how execution flow control is transferred between the object part and the logic part of an $\mathcal{OLI}$ program. Lee and Pun (Lee and Pun 1995) report the formal operational semantics of $\mathcal{OLI}$.

Suppose execution starts with a message being sent to an object. Execution follows the usual SMALLTALK style execution until a `query:` message is sent to the `logicBase` object. Control is then temporarily transferred to the logic part, where a form of generalized SLD-resolution is performed. In the execution of the logic part, SLD-resolution must be extended to handle unifications involving o-terms. Informally, the following three cases are added to the usual unification algorithm:

- When a free variable is unified with an o-term, the o-term is bound to the variable.
- When a h-term is unified with an o-term, unification fails.
- When a o-term is unified with an o-term, unification succeeds if and only if the o-terms are the same object. In other words, an o-term unifies only with itself.

During the generalized SLD-resolution, invariant messages may be sent to o-terms, in which case control will be transferred to the object part. Similar control flow pattern is exhibited when execution starts with a logical query. We should emphasize again that no state change takes place during a logical derivation. This is guaranteed by the fact that only invariant messages can be sent to o-terms during logical derivation.

6 Conclusion and Future Work

In this paper, we have justified the multiparadigm approach for software design and implementation. This multiparadigm approach serves as a better mental model than the object-oriented programming paradigm or the logic programming paradigm alone in addressing the design and implementation aspects of software development. In order to support this multiparadigm approach, we propose the $\mathcal{OLI}$ scheme to integrate the object-oriented and the logic programming paradigms at the design and the language levels. The $\mathcal{OLI}$ multiparadigm design methodology addresses multiparadigm software design by delegating the analysis and synthesis responsibilities to the object-oriented and the logic programming paradigms respectively. The $\mathcal{OLI}$ multiparadigm programming language addresses multiparadigm software implementation by providing a platform on which constructs of the object-oriented and the logic programming paradigms coexist and communicate harmoniously with each other. The $\mathcal{OLI}$ multiparadigm design methodology and programming language are complementary to each other so that the object-oriented and the logic programming paradigms play their cooperating

```
coursesTaken($tom, []).
coursesTaken($mary, [$csc110,$csc111,$art101]).
...
curCourse($tom,[$csc110,$csc111,$his179]).
curCourse($mary,[$csc234,$csc245,$art201]).
...
preReq($csc234, [$csc110,$csc111]).
...
maxLoad(1,6).  maxLoad(2,6).  maxLoad(3,5).  maxLoad(4,4).
...
degreeReq(...) :- ...

canTake(Student, Course) :-
    curCourse(Student, CC), length(CC, L), maxLoad(Student year, ML),
    L <= ML, coursesTaken(Student, CT), preReq(Course, PR),
    subset(PR, CT), Student major = Course major.

length([], 0).          subset([], _).          delete(H, [H|T], T).
length([_|T], N) :-     subset([H|T], L) :-     delete(E, [H|T], [H|T1]) :-
    length(T, N1),          delete(H, L, L1),       delete(E, T, T1).
    N is N1 + 1.            subset(T, L1).

graduationStudent(Student, Major) :-
    Student major = Major, coursesTaken(Student, CT),
    degreeReq(Major, ReqCourses), subset(ReqCourses, CT).
```

Figure 6: Code Fragments from the Logic Part of an $\mathcal{OLI}$ Program

and competent roles in software design and implementation.

Work has been started to look into the development of a CASE tool in supporting the $\mathcal{OLI}$ multiparadigm design methodology. This tool will operate on top of the $\mathcal{OLI}$ multiparadigm programming language to generate code templates automatically from a high-level design. The issues of predicate inheritance and generic predicates have not been addressed properly in the current version of the $\mathcal{OLI}$ multiparadigm language. These issues are very useful in reusing and maintaining logical predicates. Research work is starting to develop an inheritance scheme and generic mechanism which will be incorporated into the $\mathcal{OLI}$ multiparadigm language.

Acknowledgement

We thank Chong Kan Chiu for giving constructive comments to earlier drafts of the paper. This research was supported in part by a UK/HK Joint Research Scheme grant (HKG/520/8-6), funded by the British Council.

References

Aït-Kaci, H. (1984). *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures.* Ph. D. thesis, University fo Pennsylvania, Pennsylvania, Philadephia, PA, USA.

Aït-Kaci, H. (1986). An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science 45*, 293–351.

Aït-Kaci, H. and A. Podelski (1993). Towards a meaning of LIFE. *Journal of Logic Programming 16*(3 & 4), 195–234.

Booch, G. (1986). Object-oriented development. *IEEE Transactions on Software Engineering 12*(5), 211–221.

Clocksin, W. and C. Mellish (1984). *Programming in* PROLOG. Springer-Verlag.

Cox, B. (1986). *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley.

Deville, Y. (1990). *Logic Programming: Systematic Program Development*. Addison-Wesley.

Goldberg, A. and D. Robson (1989). *S*MALLTALK*-80: The Language*. Addison-Wesley.

Gullichsen, E. (1985). BiggerTalk: Object-oriented Prolog. Technical Report STP-125-85, MCC-STP, Austin, TX.

Hill, P. and J. Lloyd (1994). *The Gödel Programming Language*. The MIT Press.

Ishikawa, Y. and M. Tokoro (1986). Concurrent object-oriented knowledge representation language Orient84/K: Its features and implementation. In *OOPSLA'86*, pp. 232–241.

Janson, S. (1994). *AKL - A Multiparadigm Programming Language*. Ph. D. thesis, Computing Science Department, Uppsala University, Uppsala, Sweden.

Kowalski, R. (1979). *Logic for Problem Solving*. North Holland.

Lee, J. and P. Pun (1995). The $\mathcal{OLI}$ multiparadigm programming language and system. Technical report (in preparation), Department of Computer Science & Engineering, The Chinese University of Hong Kong.

Lloyd, J. (1987). *Foundations of Logic Programming* (Second ed.). Springer-Verlag.

McCabe, F. (1992). *Logic and Objects*. Prentice Hall.

Mellender, F. (1988). An integration of logic and object-oriented programming. *SIGPLAN Notices 23*(10), 181–185.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice Hall.

Meyer, B. (1992). *Eiffel: the Language*. Prentice Hall.

Moss, C. (1994). *P*ROLOG*++: The Power of Object-Oriented and Logic Programming*. Addison-Wesley.

Pun, P. and H. Kahn (1995). A multiparadigm programming approach for developing knowledge-based systems. In *Proceedings of the 19th IEEE Annual International Computer Software and Applications Conference*, pp. 362–368.

Shapiro, E. and A. Takeuchi (1983). Object oriented programming in concurrent prolog. *New Generation Computing 1*, 25–48.

Sterling, L. and E. Shapiro (1986). *The Art of P*ROLOG*—Advanced Programming Techniques*. The MIT Press.

Wu, S. (1991). Integrating logic and object-oriented programming. *OOPS Messager 2*, 28–37.

Zaniolo, C. (1984). Object-oriented programming in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, pp. 265–270.