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

J.H.M. Lee
Department of Computer Science & Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
Email: jlee@cse.cuhk.edu.hk
Fax: +852 2603 5024

P.K.C. Pun
Department of Computer Science
University of Manchester
Manchester, United Kingdom
Email: punpa02@cai.com

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 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. An important property of the $\mathcal{OLI}$ language is that programmers can program either in one of the paradigms alone or in a mixed paradigm without sacrificing expressiveness and efficiency. We give a formal definition of the $\mathcal{OLI}$ language and study its semantics both from the logical perspective and the object-oriented perspective. By viewing objects as an enrichment of the Herbrand universe, we define the declarative and operational semantics of $\mathcal{OLI}$. We show that $\mathcal{OLI}$'s operational semantics, a generalized form of SLD-resolution, is sound and complete. From the object-oriented point of view, the logic part of $\mathcal{OLI}$ is simply an object with logic programs as states and methods for performing logical deduction.

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 [1, 2, 3] 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 [4, 5, 6] 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.

Another motivation is the need of multiple programming paradigms in the construction of real-life software systems. A programming paradigm is a coherent conceptual framework which enables programmers to look at a domain of interest and analyze problems in this domain. Each paradigm offers a set of programming languages as representation facilities so that problem solutions can be expressed and executed on computing machines effectively. Appropriate programming paradigm for a problem domain has great impact on the programming task and the productivity of programmers. A problem of a particular domain can be solved easily using languages in one paradigm, and yet the same problem may require hours of tedious programming using languages in another paradigm. Different paradigms also facilitate different levels of programming abstraction and software design methodologies. In developing large software, we normally encounter many tasks of different domains and very often no single paradigm would be suitable for all these tasks. A multi-paradigm programming approach, which support combinations of different paradigms, provides a natural mental model for the construction of complex computer applications.

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 program 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 by the logic paradigm are clearly distinguished in the program design process. In order to achieve this, we put forward a multiparadigm 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. An important property of the $\mathcal{OLI}$ language is that it is, besides a multiparadigm language, also a logic programming language and an object-oriented language. Thus, its semantics can be studied

from two perspectives: the object-oriented view and the logical view. We present the formal syntax and semantics of the $\mathcal{OLI}$ language. The declarative semantics of $\mathcal{OLI}$ is based on viewing objects as enrichment of the Herbrand universe [7]. A generalization of SLD-resolution [7] is defined to cope with the new Herbrand terms [7]. We show that the generalized resolution principle is both sound and complete. The object-oriented semantics of $\mathcal{OLI}$ is based on the notion of a logical object, which has as states a logic program and has methods for invoking logical deduction.

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 program design level in the $\mathcal{OLI}$ scheme. A multiparadigm 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 [8] and PROLOG [9] is described in this section. We give the syntax of the $\mathcal{OLI}$ language, followed by a formal treatment of the logical and object-oriented semantics of the language. Section 6 summarizes our contributions and sheds light on future directions of research.

2 Related Work

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 following, these two emulation camps are reviewed while emphasis is placed on commenting why these two emulation strategies do not result in ideal multiparadigm programming languages and systems. This review is not intended to be exhaustive; it is limited

to what is considered representative. A more detailed treatment of this topic is reported in other literature [10, 11].

Logical Emulation of Objects There are two different approaches to emulating objects in logic programming languages and systems: objects as unconditional clauses and objects as perpetual processes.

In the first approach [12, 13], objects are represented as a set of unconditional clauses. For each of these unconditional clauses, the predicate symbol denotes a property of an object instance. The arguments of this unconditional clause state the values of this property and the reference of this object instance. All predicates of the same object reference collectively represent all instance variables and methods that object possesses. Dynamic object instance creation and instance variable update are based on the destructive modification of the logic program through non-logical constructs *e.g.* `assert` and `retract` in PROLOG. The heavy reliance on destructive constructs and the lack of encapsulation are two major disadvantages of languages and systems based on this approach. Moreover, these languages and systems force a programmer to subscribe to a clausal view of the object-oriented programming paradigm. This clasual view represents an interface which is too deep in integrating the object-oriented and the logic programming paradigms. The analysis role played by the object-oriented programming paradigm becomes redundant because all objects are subsequently converted to unconditional clauses which are relational in terms of the synthesis understanding. In this way, languages and systems based on this approach do not really support mixed paradigm programming.

In the second approach [14, 15], objects are represented as perpetual processes. These perpetual processes are perpetual predicates which continuously reinstate themselves by making recursive calls. Each of these perpetual predicates carries an argument which is a message stream identifying an object instance. Object states are represented by other arguments in the perpetual predicate. Heads of all message streams of perpetual predicates with the same predicate symbol specify all messages allowed in a certain class of objects. Object instances are created by process reduction. State update of object instances is reflected in different incarnations of perpetual processes with new state arguments through recursion. Lack of a satisfactory semantics for perpetual processes and for communication and synchronization between the processes is a major problem with languages and systems based on this approach. Moreover, this approach fails to model well the analysis role of the object-oriented programming paradigm. Although all processes can be viewed as objects, not

all objects can be naturally interpreted as processes. In this way, breaking down a problem domain in terms of processes does not necessarily agree with the analysis understanding of this problem domain which is fundamental in the object-oriented design process.

Programs written using languages and systems based on the above two approaches are usually verbose and vulnerable to subtle bugs. In order to provide a better linguistic support for object-oriented programming, a number of vanilla preprocessors, *e.g. L&O* [16] and PROLOG++ [17], have been developed. These preprocessors translate programs written in a syntax capturing the cliches used for object-oriented programming into native logic programming languages syntax. Although this approach helps to reinstate the analysis role of the object-oriented programming paradigm in the resulting languages and systems, problems with the semantics of the translated programs still remain. Moreover, the motivation of *L&O* and PROLOG++ is to extend the logic programming paradigm with features from the object-oriented paradigm in order 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 a more important objective for their integration.

Most logic programming languages and systems are based on classical logic, which is designed to be a formalism to reason about static facts and relations. It is not suitable for modeling state change occurring during object interaction in the object-oriented programming paradigm. Recently, there are a number of research proposals for non-classical logic formalisms [18, 19, 20] to describe actions and changes. However, these formalisms are usually too complicated and cumbersome to be employed as the theoretical basis of a practical programming language and system.

Object-Oriented Emulation of Predicates There are several approaches to emulating components of the logic programming paradigm with components of the object-oriented programming paradigm. Major components of the logic programming paradigm are terms, clauses and the process of unification. Major components of the object-oriented programming paradigm are objects, classes and methods. The different approaches of this emulation camp are characterized by the way they draw correspondences between their respective components.

LogiC++ [21] and Orient84/K [22] allow class methods to be specified as a set of clauses. However, there is no direct correspondence between terms in clauses and objects of classes in these languages. In this way, they only support unification of simple syntactic symbols but not complex objects defined by users. In languages such as ALF [23], PROLOG/V [24] and EXL [25, 26, 27],

clauses are represented as subclasses of a built-in class. In these languages, an object can be a term and vice versa. In this way, unification of user-defined objects is supported. However, these languages require logical predicates to be specified within classes, *i.e.* predicates do not exist independently. Such restriction implies that the object-oriented programming and the logic programming paradigms do not share an equal partnership when designing programs using these languages and systems. The TOPIC PROLOG to C++ translator [28] provides a set of built-in class constructs which allows user-supplied C++ code to manipulate and use PROLOG concepts such as queries, predicates, terms and unification. This translation strategy requires programmers to involve themselves with much low-level details of the C++ design used to implement the concepts of PROLOG. This programming burden is not relevant and should not fall upon programmers in designing and developing multiparadigm programs.

Lee and Pun [29] outline the $\mathcal{OLI}$ language as an accompaniment of the $\mathcal{OLI}$ multiparadigm program design methodology. In [30], they define the formal syntax and semantics of $\mathcal{OLI}$. This paper contains a combined and extended version of the results reported in [29, 30].

The LIFE language [31] 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 [32, 33], 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 dynamic state changes and 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; (s)he 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 a 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. For example, 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 the student 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 program design process. A set of design guidelines are then formulated based on this interface. This set of guidelines represents a multiparadigm program design methodology which integrates the object-oriented and the logic programming paradigms at the program 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 program 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, which may occur in a practical implementation.

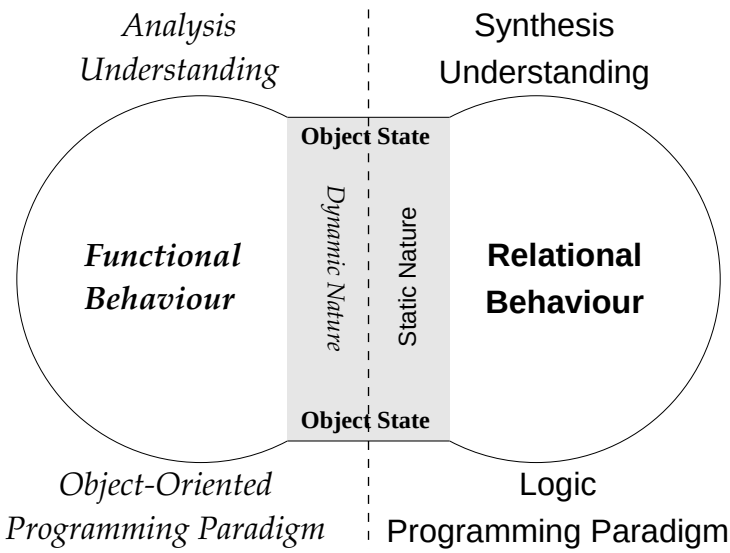

Figure 1: The $\mathcal{OLI}$ Multiparadigm Design Interface

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:

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

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 will be computed in the second case. 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

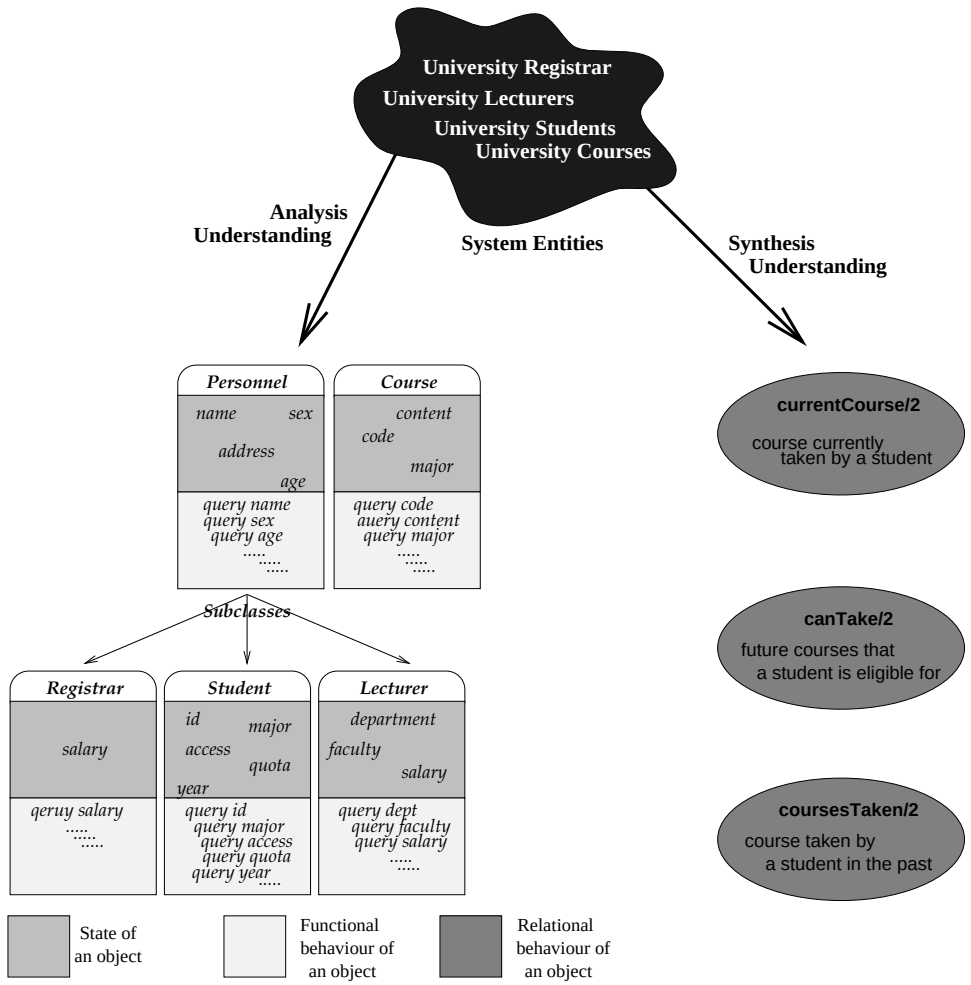

Figure 2: Decomposition of a University Registration System

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 `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 [34] and Eiffel [35].

After describing the multiparadigm language interface, we give a brief overview of the $\mathcal{OLI}$ language by illustrating 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. At appropriate places, code fragments are used to exemplify various language constructs of $\mathcal{OLI}$. Last but not least, we expose the formal syntax and semantics, both logical and object-oriented, of the language. We assume readers basic knowledge of PROLOG and SMALLTALK. Interested readers are referred to [5, 8] for detailed description of the languages.

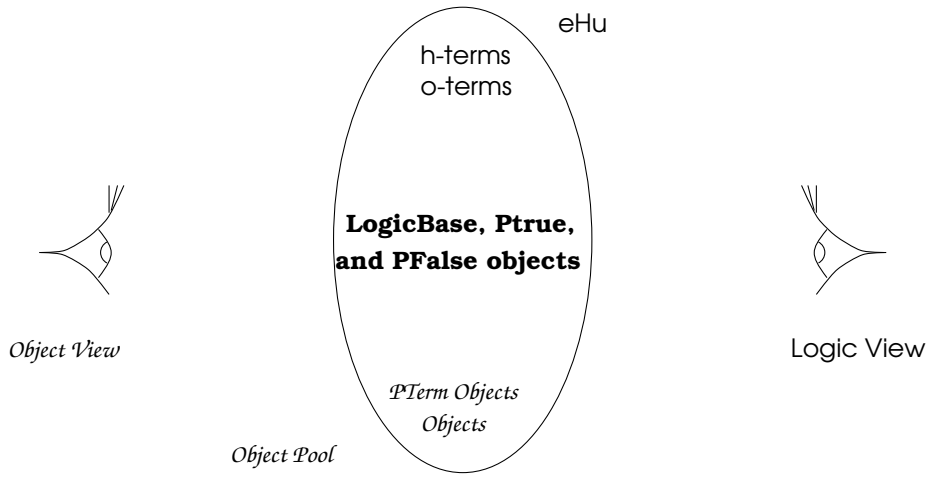

Figure 3: The $\mathcal{OLI}$ Programming Language Interface

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 [7] 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

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

instead of unification [7]. *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 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, which exhausts and acts on the list of all answer tuples using the argument block. 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 A Language Overview

In the following, we present an $\mathcal{OLI}$ implementation of a multiparadigm solution to the university registration system.

5.2.1 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. For example, the `salary` method in the `Registrar` class is for state enquiry

and the `+` message in the message expression `1+2` represents 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. One invariant instance method of a `Registrar` object inquires the salary of the object. 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 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 `curCourse/2` 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.

```
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 $\mathcal{OLI}$

```
$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 $\mathcal{OLI}$

5.2.2 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 [7] containing o-terms in addition to h-terms. The attributes 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 prerequisites 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

```
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

`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.

5.3 Program Execution

Being a multiparadigm language, $\mathcal{OLI}$ allows users to invoke its execution via either the logic part or the object part. In the $\mathcal{OLI}$ programming environment, a workspace similar to that of SMALLTALK is provided. To find out whether Mary can take the course CSC341, users can post the query "`?- canTake($mary,$csc341).`" in the workspace. To add the course CSC267 for Paul, users

Class Definition:
NameOfSuperclass **subclass:** #NameOfClass
    instanceVariableNames: 'instVarName1 ...'
    classVariableNames: 'ClassVarName1 ...'

Variant Class Methods:
Method definitions

Invariant Class Methods:
Method definitions

Variant Instance Methods:
Method definitions

Invariant Instance Methods:
Method definitions

message selector and argument names
    "comment stating purpose of message"

    |temporary variable names|
    statements

Figure 7: Templates of the Class and Method Definitions in $\mathcal{OLI}$

can try evaluating the message expression "`registrar add: $paul course: $csc267`," where `registrar` is an instance of the `Registrar` class. Mixed paradigm queries and message expressions are also possible. The query "`?- canTake($mary,Course), Course major = csc.`" inquires for computer science courses that Mary may take. The message expression

"`$logicBase query: [canTake($mary,Course),Course major = csc] ifTrue:`
`[:aCourse|registrar add: $mary course: aCourse].`"

adds an eligible computer science course for Mary. In the execution of logical queries and message expressions, execution may flow back and forth between logical deduction and message evaluation. Details of the $\mathcal{OLI}$'s operational semantics are recorded in sections 5.5 and 5.6.

5.4 Syntax

5.4.1 The Object Part

The object part of an $\mathcal{OLI}$ program consists of a set of class definitions, which formulate the behavior of the classes' object instances. The templates of $\mathcal{OLI}$ class and method definitions are shown in figure 7, where keywords are in bold. Each class must have state variable declarations and (class and instance) method definitions [8]. Each object is an *instance* of a class. A class defines a set of objects (its instances) that share the same response to a common message but these objects have different states in general. A *message* requests an operation on the part of the receiver object. It is up to the receiver to decide how to respond to the message based on

the corresponding method definition. A *message expression* describes a receiver, a selector, and possibly some arguments. The receiver and arguments are described by other expressions, whereas the selector is specified literally. We leave unspecified whether the object part should be typed and whether the object part should support class inheritance. In our current SMALLTALK-based implementation, we support class inheritance but not typing. SMALLTALK takes the view that everything, including class, is an object. We adopts this viewpoint in $\mathcal{OLI}$. Therefore, each class has *class methods* for the class itself and *instance methods* for instances of the class. Methods can be *variant* and *invariant*.

5.4.2 The Logic Part

The logic part of an $\mathcal{OLI}$ program starts with object declaration followed by a set of predicate definitions written in the style of PROLOG. Object declaration allows programmers to create and name different object instances using the classes defined in the object part. It has the form:

Name =:= Classname ObjectCreationMessage.

The objects created are called *primitive o-terms*, which have a name, a state (consisting of primitive o-terms), and a set of state-preserving invariant methods. Since primitive o-terms are instances of the classes in the object part, it possesses variant and invariant methods. *Only invariant methods are accessible by the logic part.* This is enforced by the fact that variant methods are made *invisible* to the logic part in our language implementation. An important requirement of invariant methods is that they always terminate. An *o-term* is either a primitive o-term or a message expression with an o-term receiver. The result of evaluating a message expression is always a primitive o-term, which is the *value* of the expression. For the purpose of this paper, it suffices to assume a *standard message evaluation procedure* of the SMALLTALK language.

An *alphabet* of a first-order language L with o-terms is composed of a set ν_L of variables, a set σ_L of constants, a set μ_L of o-terms, a set Σ_L of function symbols, a set Π_L of predicate symbols, a set of connectives, a set of quantifiers, and a set of punctuation symbols. O-terms also denote individuals. We are now ready to define the first-order language used in the logic part of $\mathcal{OLI}$ given by an alphabet. We need an extended notion of term [7]. A constant is a term. A variable is a term. An o-term is a term. If $\mathtt{f} \in \Sigma_L$ is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $\mathtt{f}(t_1, \ldots, t_n)$ is a term. If $\mathtt{p} \in \Pi_L$ is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $\mathtt{p}(t_1, \ldots, t_n)$ is an *atomic formula* or an *atom*. A *generalized definite program* consists of a finite

set of *generalized definite clauses* of the form $A \leftarrow B_1, \ldots, B_m$ where $m \geq 0$, A and B_i are atoms for $i = 1, \ldots, m$. A is called the *head* of the clause and $B_1, \ldots, B_m$ is the *body* of the clause. A *generalized definite goal* is a generalized definite clause without a head. The logic part of an $\mathcal{OLI}$ program is a generalized definite program. The word "generalized" can be omitted hereafter when the context is clear.

5.5 Logical Semantics

The introduction of o-terms into first-order language extends the notion of Herbrand universe [7]. We define the *enriched Herbrand universe* (**eHu**) U_L of a first-order language L to be the set of all ground terms, constructable from σ_L, μ_L, and Σ_L. *Ground terms* are terms not containing variables. Let L be the first-order language used in the logic part of $\mathcal{OLI}$. Thus the **eHu** of L is the result of augmenting the Herbrand universe of L with object instances of the classes defined in the object part. The *enriched Herbrand base* (**eHb**) B_L of a first-order language L is the set of all ground atoms, constructable from Π_L and U_L. The *Herbrand interpretation* for L is given by:

- The domain of the interpretation is U_L.
- Constants $c \in \nu_L$ and o-terms $o \in \mu_L$ are assigned themselves in U_L.
- If $\mathtt{f} \in \Sigma_L$ is an n-ary function symbol, then $\mathtt{f}$ is assigned the mapping from $(U_L)^n$ into U_L defined by $(t_1, \ldots, t_n) \rightarrow \mathtt{f}(t_1, \ldots, t_n)$.
- If $\mathtt{p} \in \Pi_L$ is an n-ary predicate symbol, then $\mathtt{p}$ is assigned a mapping from $(U_L)^n$ into $\{\mathtt{true}, \mathtt{false}\}$.

The meaning of the connectives, quantifiers, Herbrand model, satisfiability and logical consequences can now be defined as usual. Without loss of generality, we assume that the underlying first-order language of a set of generalized definite program P is defined by the constants, o-terms, function symbols and predicate symbols appearing in P. Therefore, it makes sense to refer to the **eHu** U_P and **eHb** B_P of a program P and also refer to the Herbrand interpretations of P as subsets of B_P.

5.5.1 Declarative Semantics

Invariant methods of an object are for state enquiry only. We restrict that invariant methods must (1) always terminate and (2) embody a function abstraction and a deterministic computation. Therefore, messages to a primitive o-term must be evaluated to another primitive o-term. We

define the *value* of an o-term as follows. The value of a primitive o-term t is t itself. The value of a non-primitive o-term (a message expression with an o-term receiver) e is the result of the evaluation of e.

The values of o-terms denote additional individuals, extending the Herbrand universe. *The restriction of only state-preserving messages to o-terms ensure the static nature of the enriched Herbrand universe throughout a logical computation.* This is an essential property for the following results to hold.

Proposition 5.1: (Model Intersection Property) If P is a generalized definite program and $\{M_i\}$ is a non-empty set of Herbrand models for P, then $M_P = \cap\{M_i\}$, called the *least Herbrand model*, is a Herbrand model for P.

Proof: Obviously, M_P is a Herbrand interpretation for P. Suppose M_P is not a Herbrand model of P. There exists a ground instance $C\theta$ of a clause $C(= A \leftarrow B_1, \ldots, B_n (n \geq 0))$ in P so that $C\theta$ is false under M_P. It follows that $\{B_1, \ldots, B_n\} \subseteq M_P$ and $A \notin M_P$. Thus, there exists M_i so that $\{B_1, \ldots, B_n\} \subseteq M_i$ and $A \notin M_i$, which implies that $C\theta$ is false under M_i. This is a contradiction since M_i is assumed to be a Herbrand model of P. ∎

Theorem 5.2: If P is a generalized definite program, then

$$M_P = \{A \in B_P \mid A \text{ is a logical consequence of } P\}.$$

Proof: The proof of the theorem is completed by the following chain of equivalence.

$$\begin{aligned}
& A \text{ is a logical consequence of } P \\
\Leftrightarrow\ & P \cup \{\neg A\} \text{ is unsatisfiable [7]} \\
\Leftrightarrow\ & P \cup \{\neg A\} \text{ has no Herbrand models [7]} \\
\Leftrightarrow\ & \neg A \text{ is false in all Herbrand models of } P \\
\Leftrightarrow\ & A \text{ is true in all Herbrand models of } P \\
\Leftrightarrow\ & A \in M_P.
\end{aligned}$$

∎

5.5.2 Operational Semantics

Unification is an important step in the execution of logic programs. In unifying two expressions containing o-terms, we first replace each non-primitive o-term in the expressions with its value by

evaluating the o-term (a message expression), which, by definition, invokes only state-preserving invariant methods. Therefore, the notion of substitution must be extended to include primitive o-terms. A *substitution* is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where v_i's are variables, each t_i is a term distinct from v_i, each t_i cannot be a non-primitive o-term, and the variables $v_1, \ldots, v_n$ are distinct. The definitions of unification and most general unifier, and the unification algorithm can remain the same by treating o-terms as constants. In other words, a primitive o-term unifies only with itself. Note that o-terms denote objects with state. The same object must have the same state.

Execution of generalized definite programs is based on a generalization of SLD-resolution [7]. Let P be a generalized definite program and $G_i = \leftarrow A_1, \ldots, A_j, \ldots, A_n$ a definite goal, where $n > 0$. The goal $G_{i+1} = \leftarrow (A_1, \ldots, A_{j-1}, B_1, \ldots, B_m, A_{j+1}, \ldots, A_n)\theta$ is *derived* from G_i using *input* definite clause c and most general unifier θ if

1. if A_j, the selected atom in G_i, contains non-primitive o-terms (*i.e.* message expression), then each message expressions is replaced by its value,

2. $c = A \leftarrow B_1, \ldots, B_m$ is a suitable variant [7] of a clause in P and θ is the most general unifier of A_j and A.

A *derivation* is a, possibly infinite, sequence of definite goals $G = G_0, G_1, G_2, \ldots$ such that G_{i+1} is derived from G_i generating substitution θ_{i+1}. A derivation is *successful* if it is finite and the last goal contains no atoms. The *computed answer* of a successful derivation of length m is the composition $\theta_1\theta_2 \cdots \theta_m$ of the most general unifiers generated in each step of the derivation. A derivation is *finitely failed* if it is finite, ends in a non-empty goal and condition 2 does not hold.

The evaluation of non-primitive o-terms during a derivation step is deterministic and provides a channel to invoke object computation during logical inference.

5.5.3 Soundness and Completeness

A generalized definite program asserts what is true about the intended relations among individual in the **eHu**. Since only state-preserving messages can be sent to o-terms during derivation, the **eHu** is static. In addition, evaluation of non-primitive o-terms always terminate. Thus it is easy to check that the derivation procedure for generalized definite program is both sound and complete.

Theorem 5.3: (Soundness) Let P be a generalized definite program. If a definite goal $\leftarrow G$ has a successful derivation with computed answer θ, then $P \models \forall(G\theta)$, where $\forall(F)$ is the universal closure

of the formula F.

Proof: Let $\leftarrow G = \leftarrow A_1, \ldots, A_k$ and $\theta_1, \ldots, \theta_n$ be the sequence of most general unifiers used in a successful derivation of $P \cup \{\leftarrow G\}$. We prove that $\forall((A_1 \wedge \cdots \wedge A_k)\theta_1 \cdots \theta_n)$ is a logical consequence of P by induction on the length of the successful derivation.

Suppose $n = 1$. Then $\leftarrow G$ must be of the form $\leftarrow A_1$ and P has a unit clause of the form $A \leftarrow$ so that $A_1\theta_1 = A\theta_1$. Since $A_1\theta_1 \leftarrow$ is an instance of a unit clause in P, $\forall(A_1\theta_1)$ is a logical consequence of P.

Suppose the result holds true for successful derivation of length $n-1$. Suppose $\theta_1, \ldots, \theta_n$ is the sequence of most general unifiers used in a successful derivation of length n. Let $A \leftarrow B_1, \ldots, B_q (q \geq 0)$ be the first input clause and A_m the selected atom of $\leftarrow G$. By the induction hypothesis, $\forall((A_1 \wedge \cdots \wedge A_{m-1} \wedge B_1 \wedge \cdots \wedge B_q \wedge A_{m+1} \wedge \cdots \wedge A_k)\theta_1 \cdots \theta_n)$ is a logical consequence of P. Thus, if $q > 0$, $\forall((B_1 \wedge \cdots \wedge B_q)\theta_1 \cdots \theta_n)$ is a logical consequence of P. Consequently, $\forall(A_m\theta_1 \cdots \theta_n)$, which is the same as $\forall(A\theta_1 \cdots \theta_n)$, is a logical consequence of P. Hence $\forall((A_1 \wedge \cdots \wedge A_k)\theta_1 \cdots \theta_n)$ is a logical consequence of P. ∎

To prove the completeness result, we need the following lemmas, which are straight-forward adaptation from [7].

Lemma 5.4: Let P be a generalized definite program and A an atom. Suppose $\forall(A)$ is a logical consequence of P. Then there exists a successful derivation of $P \cup \{\leftarrow A\}$ with the identity substitution as the computed answer. ∎

Lemma 5.5: (Lifting Lemma) Let P be a generalized definite program, $\leftarrow G$ a definite goal and θ a substitution. Suppose there exists a successful derivation of $P \cup \{\leftarrow G\theta\}$. Then there exists a successful derivation of $P \cup \{\leftarrow G\}$ of the same length such that, if $\theta_1, \ldots, \theta_n$ are the most general unifiers from the successful derivation of $P \cup \{\leftarrow G\theta\}$ and $\theta'_1, \ldots, \theta'_n$ are the most general unifiers from the successful derivation of $P \cup \{\leftarrow G\}$, then there exists a substitution γ such that $\theta\theta_1 \cdots \theta_n = \theta'_1 \cdots \theta'_n\gamma$. ∎

Theorem 5.6: (Completeness) Let P be a generalized definite program and $\leftarrow G$ a definite goal. If there exists a substitution θ such that $P \models \forall(G\theta)$, then there exists a successful derivation for $\leftarrow G$ with computed answer σ and a substitution γ such that $\theta = \sigma\gamma$.

Proof: Let $\leftarrow G = \leftarrow A_1, \ldots, A_k$. By lemma 5.4, there exists a successful derivation of $P \cup \{\leftarrow A_i\theta\}$

such that the computed answer is the identity substitution for each $i = 1, \ldots, k$. We can combine these successful derivations into a successful derivation of $P \cup \{\leftarrow G\theta\}$ such that the computed answer is the identity substitution.

Suppose the sequence of most general unifiers of the successful derivation of $P \cup \{\leftarrow G\theta\}$ is $\theta_1 \cdots \theta_n$. Then $G\theta\theta_1 \cdots \theta_n = G\theta$. By lemma 5.5, there exists a successful derivation of $P \cup \{\leftarrow G\}$ with most general unifiers $\theta'_1 \cdots \theta'_n$ such that $\theta\theta_1 \cdots \theta_n = \theta'_1 \cdots \theta'_n\gamma'$, for some substitution γ'. Let σ be $\theta'_1 \cdots \theta'_n$ restricted to the variables in G. Then $\theta = \sigma\gamma$, where γ is an appropriate restriction of γ'. ∎

5.6 Object-Oriented Semantics

Formal semantics of practical object-oriented languages is an unsettled issue. In the following, we give an informal account of the semantics of the interface between the logic part and the object part as viewed from the object part. Description below is based on standard SMALLTALK semantics.

Execution of an $\mathcal{OLI}$ program can be invoked from the object part by sending a message to an object. In this case, the *outermost* computation is an object-oriented computation. Execution follows the usual SMALLTALK style execution until a `query:` message is sent to the `$logicBase` object. Control is then temporarily transferred to the logic part, where generalized SLD-resolution is performed. We call this a *sub-computation*, which is logical in this case. Control is returned to the object part after the sub-computation is finished. Execution continues in an object-oriented style until program termination or another `query:` message is sent to the `$logicBase` object. Objects can change states before and after a logical sub-computation. During a logical sub-computation (in the form of a generalized SLD-derivation), invariant messages may be sent to o-terms, in which case control will be transferred to the object part. This is an object-oriented sub-computation embedded in a logical computation. No state change can take place, which is guaranteed by the fact that only invariant methods can be invoked.

Similar control flow pattern is exhibited when an $\mathcal{OLI}$ program execution is initiated by a logical query. The outermost computation is logical. Control may be transferred to the object part when messages are sent to o-terms. However, these object-oriented sub-computation are all embedded in a logical computation. Therefore, no state updates will take place.

In summary, $\mathcal{OLI}$ is designed in such a way that no state updates occur in a logical computation or any object-oriented sub-computation embedded in a logical computation. In other words, state updates can occur only in an outermost object-oriented computation. When object state changes

between two different logical sub-computations during the execution of an outermost object-oriented computation, each logical derivation corresponds to a different static **eHu**.

6 Concluding Remarks

In summary, 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 and competent roles in software design and implementation.

The $\mathcal{OLI}$ language is carefully designed and engineered so as to combine the power of object-orientation and logical deduction. Programmers can choose to program entirely in any one of the paradigms without loss in expressiveness, execution efficiency, and complication in semantics. Programming in the mixed paradigm, however, provides extra expressiveness not available in either paradigm. We give the formal syntax and semantics of $\mathcal{OLI}$. Soundness and completeness result is established, retaining the simple and elegant semantics of logic programming. An important characteristics of $\mathcal{OLI}$ is that an $\mathcal{OLI}$ computation makes sense in the context of both paradigms. From the object-oriented point of view, an $\mathcal{OLI}$ computation is a sequence of object interactions via messages. Logic computation is just a message to the `$logicBase` built-in object. From the logical point of view, an $\mathcal{OLI}$ computation is a generalized SLD-derivation. Object computation merely inquires the states of o-terms or performs function abstraction. We believe that our work provides a good reference point for foreign language interface designers for either PROLOG or SMALLTALK.

The following open problems remain to be answered. First, we observe that certain meta-programming and arithmetic facilities in PROLOG can now be explained as object computation.

For example, the goal "`X=1+2`" can be viewed as unifying variable "`X`" with the non-primitive o-term "`1+2`." It is interesting to study the meta-programming implications in depth. Second, the built-in object `$logicBase` encapsulates the logic part of an $\mathcal{OLI}$ program. By allowing more than one `$logicBase` object, we may get a module system for the logic part of $\mathcal{OLI}$ for free. Third, objects are inherently typed. It is interesting to investigate if typing can be introduced into the logic part of $\mathcal{OLI}$, resulting in an inheritance scheme and generic mechanism for predicates. These issues are very useful in reusing and maintaining logical predicates. For this, we need to generalize the notion of o-terms. The ψ-calculus [32, 33] would be a good starting point. Fourth, a rudimentary implementation of the $\mathcal{OLI}$ language has been constructed [36] but more work is needed for an efficient implementation of the $\mathcal{OLI}$ language. A programming environment similar to that of SMALLTALK should also be investigated. Fifth, a CASE tool is needed to support the $\mathcal{OLI}$ 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.

Acknowledgement

This research was supported in part by a UK/HK Joint Research Scheme grant (HKG/520/8-6) funded by the British Council, and by a Direct Grant of The Chinese University of Hong Kong.

References

[1] B.J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1986.

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

[3] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[4] R.A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, August 1979.

[5] L. Sterling and E. Shapiro. *The Art of Prolog—Advanced Programming Techniques.* The MIT Press, 1986.

[6] Y. Deville. *Logic Programming: Systematic Program Development.* Addison-Wesley, 1990.

[7] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[8] A. Goldberg and D. Robson. *SMALLTALK-80: The Language.* Addison-Wesley, 1989.

[9] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1984.

[10] A. Davison. A survey of logic programming-based object oriented languages. Technical Report 92/3, Department of Computer Science, University of Melbourne, 1992.

[11] V. Alexiev. Mutable object state for object-oriented logic programming: A survey. Technical Report TR93-15, Department of Computer Science, University of Alberta, 1993.

[12] C. Zaniolo. Object-oriented programming in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 265–270, 1984.

[13] E. Gullichsen. BiggerTalk: Object-oriented Prolog. Technical Report STP-125-85, MCC-STP, Austin, TX, 1985.

[14] E. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing*, 1:25–48, 1983.

[15] S. Janson. *AKL - A Multiparadigm Programming Language.* PhD thesis, Computing Science Department, Uppsala University, Uppsala, Sweden, 1994.

[16] F.G. McCabe. *Logic and Objects.* Prentice Hall, 1992.

[17] C.D.S. Moss. *Prolog++: The Power of Object-Oriented and Logic Programming.* Addison-Wesley, 1994.

[18] V. Alexiev. Applications of linear logic to computation: An overview. Technical Report TR93-18, Department of Computer Science, University of Alberta, 1993.

[19] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 93/06, Department of Computer Science, SUNY at Stony Brook, 1993.

[20] P. Tarau, V. Dahl, and A. Fall. Bactrackable state with linear assumptions, continuations and hidden accumulator grammars. Technical report, Logic Programming Group and Computing Sciences Department, Simon Fraser University, 1995.

[21] S. Wu. Integrating logic and object-oriented programming. *OOPS Messager*, 2(1):28–37, 1991.

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

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

[24] Digitalk Inc. *Prolog/V Documentation with Smalltalk/V Mac*, 1988.

[25] P.K.C. Pun. Design of EXL - a multiparadigm programming language for developing knowledge-based applications. RC3 Research Project Report RC3/P02, Department of Computing Science, University of Manchester, 1991.

[26] P.K.C. Pun. Implementation issues of EXL. RC3 Research Project Report RC3/P03, Department of Computing Science, University of Manchester, 1992.

[27] P.K.C. Pun. A multiparadigm programming approach for developing knowledge-based systems. In *Proceedings of the 19th IEEE Annual International Computer Software and Applications Conference*, pages 362–368, 1995.

[28] R.N. Horspool and M.R. Levy. Translator-based multiparadigm programming. *Journal of Systems & Software*, 23(1):39–49, 1993.

[29] J.H.M. Lee and P.K.C. Pun. $\mathcal{O}$bject $\mathcal{L}$ogic $\mathcal{I}$ntegration: a scheme for multiparadigm design and programming. In *Proceedings of the Eighteenth International and Exhibition Technology of Object-Oriented Languages and Systems*, pages 265–278, Melbourne, Australia, November 1995.

[30] J.H.M. Lee and P.K.C. Pun. An overview of the $\mathcal{OLI}$ multiparadigm programming language and its semantics. In *Proceedings of the Third International Conference on Object-Oriented Information Systems*, pages 79–92, 1996.

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

[32] H. Aït-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures.* PhD thesis, University fo Pennsylvania, Pennsylvania, Philadephia, PA, USA, 1984.

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

[34] P. Hill and J. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.

[35] B. Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[36] J.H.M. Lee and P.K.C. Pun. The $\mathcal{OLI}$ multiparadigm programming language and system. Technical report (in preparation), Department of Computer Science and Engineering, The Chinese University of Hong Kong, 1997.