Extending Eiffel as a Full Life-cycle Language

Alonso J. Peralta
Joan Serras

Report LSI–96–45–R

Extending Eiffel as a Full Life-cycle Language

Alonso J. Peralta, Joan Serras

Abstract

One of the Object Technology goals is to offer a seamless transition in the development of systems. One way to ensure this seamless transition is using the same language throughout the process in order to avoid the need for translations between phases or activities. In this paper we present an extension to Eiffel which we propose as a full life-cycle language. This language is capable of being a "lingua franca" between different analysis and design methodologies and their notations and the CASE tools which support them.

1. Introduction

Our aim is to define a full life-cycle language, that is to say, a language which enables us to carry out the different tasks needed to specify, analyze, design, implement and document object-oriented systems. In order to create such a language, first of all we must define very clearly what this language must be able to describe. We have compared 16 analysis and design methodology notations, Objectory (or OOSE) ([Ja92]), Shlaer/Mellor ([Sh88], [Sh91]), Booch ([Bo91], [Bo93]), Coad/Yourdon ([Co91a], [Co91b], [Co93]), Martin/Odell ([Ma92], [Ma94], [Ma93]), Fusion ([Cl94]), Syntropy ([Ck94]), Hood ([Ro92]), Bon ([Ne94]), OMT ([Ru91]), Moses ([He94a]), RDD ([Wi90]), Kiss ([Kr94]) and ADM4 ([Fi93]), SOMA ([Gr94b]), UMT ([BoRu96]) so that we can see what they are able to describe and what they have all in common. The resultant common core is that which we shall incorporate into the full life-cycle language: relations between objects, the description of supra-class constructs, states and transitions to describe the dynamic behavior of objects, description of use-cases and incrementing the specifying and documenting qualities of Eiffel.

To develop a full life-cycle language we have two possibilities, either we define a complete new language or we extend an existing one. We have rejected the first

alternative because of the difficulty that represents introducing a new language to the computing community. On the other hand, in contrast with what has happened extending non-OO languages, such as C or COBOL, with OO concepts, there are no backdraws in extending a pure OO programming language with analysis and design constructs. The reason of this is that there are no contradictions between OO programming concepts and OO analysis/design concepts, and therefore the augmented language is consistent with itself. This non-existence of contradictions is the result of the OO seamless transition, since the main task for all activities (analysis, design, programming) is the identification and definition of objects and their relations.

After a careful comparison of 8 object-oriented languages (OO-COBOL, Object Pascal, Objective-C, C++, Simula, Smalltalk, Eiffel and Sather), we have concluded that the most appropriate language, as far as the possibilities offered by OO are concerned, is Eiffel. It is a pure OO language and, therefore, it does not carry any servitudes from a previous procedural language. Modeling must exclusively be worked out by means of OO concepts. Of these languages it is the only language which includes formal methods. It is, after C++ and Smalltalk, the most widespread language, both in industry and in university. And finally, it is one of the easiest languages to use.

Therefore, instead of creating a new language, we have taken Eiffel ([Me92]) as a basis, and we have extended it by adding new structures. We have called this new language Eiffel+ (for the purpose of this paper) and a compiler translating Eiffel+ to Eiffel has been built.

The description we will carry out will be extending Eiffel's syntax and we assume a familiarity of the reader with Eiffel.

2. Description of Eiffel+

2.1 Relations

Object relations are treated in all object-oriented analysis and design methodologies in order to describe the structural characteristics of a problem domain. Most methodologies have typified the possible relations (non-inheritance) at the analysis

level into one of the following types: aggregation, association, use-of. And for each of these types of relations they have broken them down into various sub-types. Therefore, a high-level language which can support the complete life-cycle must be able to allow the description of the relations that may be used and how must they be transformed into client-server relationships, which aside from inheritance is the only supported by OOPL. Interesting is that fact that most OO-CASE tools support some sort of template to describe relations separately from the classes. These templates are used afterwards to generate code for the relation ([Pe94b]). We would also reference Mahesh Dodani ([Do95]) with his proposal of supporting interactions as first-class citizens in object oriented methodologies.

All the OO methodologies propose modeling relations in some kind of so-called static chart. This chart is intended for two kind of people: the user and the designer (as opposed to analyst) and/or programmer. The use of the chart in each case is very different. With regard to the user, we want to describe his/her world: existing entities and the relations between them. These relations, as we have already said, can be of three types: aggregation, association, use. However, as we have already mentioned, each of these can, at the same time, be of several sub-types. For instance, Odell describes six kinds of compositions (aggregations) ([Od94a]): Component-Integral Object, Material-Object, Part/Portion-Object, Place-Area, Member-Bunch, Member-Partnership

Each of these compositions has different semantics and, therefore, different consequences for design and programming. Although there are subtle differences between these compositions, they are not usually significant in our communication with the user.

However, the second person the static chart of objects is intended for, the designer and/or programmer, needs to know exactly what the analyst means in order to be able to make a correct translation into the client-server model supported by current programming languages.

The decisions the designer/programmer will have to take are as follows: the attributes to be added to each member of the relation, the methods to be added to each member of the relation, the description of the objects created by the relation, the delegation of services between the members, the external visibility of the members of the relation, the cardinality between members and the way of

implementing it, the constraints imposed on the relation in the members' subclasses, the location of the members' identifiers for solving persistance or access between members.

Unfortunately, up to now, there is no precise description accepted for the different types of relations. Therefore, only a textual extension of the relation's semantics, carried out by the analyst, will allow us to clear up how to implement a relation. Any textual description is liable to cause confusion and communication problems.

In other words, we miss a formal description of the relations' semantics. As a programming language is a formal tool of description, what we propose is to allow the analyst to describe, via an entity **relation**, what is understood by each type of relation that is used in an analysis.

Our description of relation is, expressed in BNF syntax:

```
RELATION_DECLARATION: relation RELATION_NAME [FORMAL_GENERICS]
                      CLASS_DECLARATIONS
                      end [-- end relation RELATION_NAME]
RELATION_NAME: identifier
FORMAL_GENERICS: "["FORMAL_GENERIC_LIST"]"
FORMAL_GENERIC_LIST: {FORMAL_GENRIC","...}
FORMAL_GENERIC: FORMAL_GENERIC_NAME [CONSTRAINT]
              | routine FORMAL_ROUTINE_IDENTIFIER
FORMAL_GENERIC_NAME: identifier
CONSTRAINT: "->" CLASS_TYPE
FORMAL_ROUTINE_IDENTIFIER: identifier

RELATION _DEFINITION: relation RELATION_NAME is
                                RELATION_NAME [REAL_GENERICS]
REAL_GENERICS: "["REAL_GENERIC_LIST "]"
REAL_GENERIC_LIST: {REAL_GENERIC ";"....}
REAL_GENERIC: REALGENERIC_TYPE
            | REAL_GENERIC_PROCEDURE
REAL_GENERIC_TYPE: FORMAL_GENERIC_NAME ":" REAL_GENERIC_MEMBERS
REAL_GENERIC_MEMBERS: {REAL_GENERIC_NAME","....}
REAL_GENERIC_PROCEDURE: routine FORMAL_ROUTINE_IDENTIFIER ":"
                        REAL_GENERIC_MEMBERS
```

As can be seen there are two separate descriptions. The first is the declaration, that which would correspond to the template of an abstract generic relation. The second is the definition, which will correspond to the "instantiation" of the relation. The declaration has all the basic mechanisms for enrichment of participant classes and their translation to client/server relations. The definition adapts the general declaration to the particular needs of each real relation.

However, since there are different relation types that will be repeated in each new analysis we make, it would be interesting to be able to create a catalog of abstract/generic relations (what we call relation declarations) in which the basic schema is described and afterwards we would only require to concretize each application's particular aspects.

This idea of a catalog of abstract/generic relations has recently been further developed in what has been called patterns. We have been able to apply our relation to all the patterns described in [Ga94] and have created a library of instantiable

Our definition of relations in Eiffel+ allows also the inheritance between relations, but has not been included for simplicity.

2.2 States

Most object-oriented analysis and design methodologies incorporate the concept of an object's states in order to explain in a clear way the dynamic behavior of the objects that form the system. Therefore, a high-level language which can support the complete life-cycle must be able to allow the description of states and of the dynamic behavior of the system. The textual notation we have used gives support to the graphical notation created by Harel ([Ha87]), which we believe has become the main paradigm used by most methodologists. We would also like to refer to ([So95]) where a proposal is made to integrate state inheritance and objects in a dynamic model extension.

When the object is in a state, the object's behavior is univocal. Therefore, states describe the dynamic behavior of an object, for they specify which is the object's behavior when the object is in each of the states in question. That is to say, a state describes how the same object reacts before different stimuli according to the state in which the object is. Or, in other words, the differences between classes of states lie in their behavior.

An object can go through different states. By this we mean that the object, during its life-cycle, can be in different states. However, it can be in only one state at a given time. The change of state is brought about by the stimuli it reacts to.

We can consider three types of states in time. Initial, the valid states at the moment of creation of an object. Transit, the non-initial states which have transitions to other states. Final, the states from which there is no transition to other states. If the action related to one of these states is a destructor, then we shall say that this is a terminal state.

An event is what is caused or occurs in a system. An event can affect one or several objects. An event is identified by a name and the parameters or additional information which is involved in the event. Any event causes a reaction within the system, which is what we call action.

From the point of view of the application, an action is the code (method) which is associated with the event, where the event gives a name to the method and also determines its parameters.

Not all events are the same. Basically, authors make the following distinction. Events which are external to the system: They are the ones caused by the user (the user presses a button or turns off a switch) or by actual mechanophysical systems (the water tank is full and the device for turning off the inlet valve is activated). Events which are internal to the system: The computer system of a bank detects an attempt to draw money by a card which has been reported stolen and it causes the event resulting in the withholding of the card.

According to their complexity, we can distinguish between. Simple events: those which do not cause or generate any other event. Compound events: those which cause a sequence of events (within this group we shall nearly always include external events).

A transition is the change of state. If a given object is in a state, then a certain change of state or transition can be brought about by the occurrence of an acceptable event for this state.

We have extended Eiffel with states in the following form:

```
STATE_DECLARATION: state MACHINE_STATES_DEF
MACHINE_STATES_DEF: MACHINE_STATES
                  | MACHINE_STATES_LIST
```

```
MACHINE_STATES_LIST: machine identifier MACHINE_STATES
                   | MACHINE_STATES_LIST machine identifier
                     MACHINE_STATES
MACHINE_STATES: BLOC_STATES_LIST ALLOW_CLAUSE
BLOC_STATES_LIST: BLOC_STATES
                | BLOC_STATES_LIST BLOC_STATES
BLOC_STATES: STATES_QUALIFIER STATES_LIST
STATES_QUALIFIER:
                   | initial
                   | transit
                   | final
STATES_LIST: STATE_ITEM
           | STATE_LIST ";" STATE_ITEM
STATE_ITEM: STATE_DEF CREATE_CLAUSE_OPT EVENT_STATEMENT_OPT
STATE_DEF: identifier
         | identifier ":" EXPRESSION
           | identifier nest to MACHINE_STATES end
EVENT_STATEMENT_OPT:
                         | EVENT_STATEMENT
EVENT_STATEMENT: event EVENT_ACTION_LIST
EVENT_ACTION_LIST: EVENT_ACTION_CLAUSE
                  |EVENT_ACTION_LIST ";" EVENT_ACTION_CLAUSE
EVENT_ACTION_CLAUSE: EVENT_ACTION TRANSITION_STATEMENT_OPT
EVENT_ACTION: identifier | routine | call
TRANSITION_STATEMENT_OPT:
                                  | TRANSITION_STATEMENT
TRANSITION_STATEMENT: transition to DESTINATION_BLOC
DESTINATION_BLOC: STATE_NAME
                 |DESTINATION_WITH_GUARDS "," STATE_NAME when
EXPRESSION
ALLOW_CLAUSE: allow EVENT_LIST";"
EVENT_LIST: identifier
          | EVENT_LIST "," identifier
CREATE_CLAUSE_OPT:
                        |CREATE_CLAUSE
CREATE_CLAUSE: created by identifier
```

Our definition of states in Eiffel+ allows also the description of orthogonal states, nested states and state inheritance between classes (which has not been included for simplicity).

2.3 Clusters

A cluster is a language entity which enables us to encapsulate other language entities. Therefore, a cluster encapsulates class definitions, declarations of objects, definitions and/or declarations of relations and definitions of routines. However, given the fact that a cluster is also an language entity, a cluster can encapsulate other clusters, though this does not mean that a cluster can encapsulate itself.

From the point of view of analysis, all the entities which form a cluster must have and keep one single, logical and coherent concept which reflects a common task that the cluster's constituents must provide.

Among the many reasons and arguments in favor of a supra-class construct the following are those that have made us incorporate the notion of cluster into the language: it allows us to group language entities into units of higher encapsulation than that of class so that it will enable us to manipulate and control more easily large systems; it allows us to encapsulate methods or routines which cannot be easily associated with a class (traversals), by means of those entities to which it is related to a certain extent; it allows us to use libraries in a more rational way, for it enables us to select those components which are really needed instead of incorporating the whole cluster. It makes library searching much easier since it reduces by two orders of magnitude it's size. This facility clusters provide is closely connected to the notion of kit introduced by Berard ([Be93]); it is the supporting tool for the construction of domains.

Eiffel+ is not the first language to support both classes and some form of higher construct. The following object-oriented languages support the concept of cluster: TurboPascal (Unit), Modula-3, Modula-90 and Ada (package). Object-oriented languages which do not support the concept of cluster are Eiffel (except for LACE), Sather, C++, Smalltalk and Simula.

As mentioned earlier, clusters in Eiffel+ enable us to define entities of higher encapsulation than that of class, thus coinciding with the concept of subsystem. Therefore, clusters allow us to group classes and other language entities (objects, relations, constants, clusters, ...) which all together form a logical and coherent entity within the global system.

Declaration of clusters in Eiffel+

A cluster consists of a first part which indicates the relations between this cluster and either other clusters from the system or a library of clusters. Relations between clusters can be of three kinds:

Inherit. We can define a cluster through it's differences from another cluster. The cluster inherits all other clusters' entities. For each cluster it inherits from we can specify if we are redefining or renaming any of the entities of the parent cluster, just in

the same way as we do when a class inherits features from some other class. As done with a class's features, we can also change the visibility by using the key word **export**.

Import. A cluster may import specified entities from a supplier-cluster. This mechanism is especially useful for reusing components defined in other clusters without needing to incorporate the whole cluster into the application.

Use. To describe the relations among clusters of a system. The client cluster uses the definitions and declarations of the entities from the cluster server.

Clusters can have methods of initialization, which are useful for initializing either variable or constant objects from the cluster or other appropriate operations of initialization. These initializing routines are defined as any other routine in the cluster is, even though we must specify its name right after the key word creation within the definition of cluster.

Next, we must declare and define the body or main part of the cluster. After the key word features we shall define the entities that form the cluster. Here it is where classes, objects, routines and relations are defined as explained in earlier sections. Finally, we may declare the cluster's **invariant**.

Our description of clusters in Eiffel+ is:

```
CLUSTER: cluster identifier
                 INHERIT_CLAUSE
                 IMPORT_CLAUSE
                 USE_CLAUSE
                 CREATION_CLAUSE
                 FEATURE_BLOCS
                 INVARIANT_CLAUSE
          end
INHERIT_CLAUSE: [inherit LIST_OF_CLUSTERS
                 [rename identifier as identifier]
                 [redefine LIST_OF_IDENTIFIERS]]
IMPORT_CLAUSE:  [import LIST_OF_ENTITIES from identifier]
USE_CLAUSE:      [use LIST_OF_CLUSTERS
                 [rename identifier as identifier]]
CREATION_CLAUSE: creation LIST_OF_ROUTINES
FEATURE_BLOCS:{FEATURE_BLOC,.....}
FEATURE_BLOC: feature CLIENTS
              USE_CLAUSE
              FEATURE_DECLARATION_LIST
INVARIANT_CLAUSE: [invariant  ASSERTION]
end --cluster
```

2.4 Use-cases

In an implemented object-oriented system both the dynamic behavior of objects and the interaction (messages) between objects is fully described within the code of methods themselves. During analysis or design, it is very often convenient to be able to describe the system's expected dynamic behavior for a specific functionality. This descriptive task allows us to state precisely which objects are involved in the given process, which methods are being used and in which order or sequence the messages are being sent. Therefore, describing a specific or particular functionality enables us to determine which methods are necessary, which class these methods correspond to and finally, it also allows us to carry out a test of the communication or flow within the application.

Whereas in structured methodologies the system's functionality is described by use of data flow diagrams (DFD), object-oriented methodologies use basically another concept, known as interaction diagrams [Ja92 [Bo93] or Event Trace Diagrams [Ru91] or Walk-throughs [Wi90] or Message trace diagrams/Object message diagram [BoRu96]. These interaction diagrams are used to describe what is called usecases [Ja92], or scenarios [Bo93], [Ru91], [Wi90].

Usecases are useful for several reasons: For determining the system's functionality. For determining the sequence of interactions between the actors and the system. They also serve as a basis for writing the system's manuals (since usecases describe the interaction between the user and the system but not the internal evolution, they are described in the interaction diagrams).

On the other hand interaction diagrams are used: For stating precisely how the system solves a given functionality for a certain usecase: (1) by identifying the participant objects, (2) by identifying the methods or responsibilities for each one of the objects, and (3) by identifying the flow or order of interaction between the objects. For distributing responsibilities among the objects (distribute to those objects to which they really correspond, for this is not a random distribution). For testing and checking the design of the diagrams of objects.

Usecases in Eiffel+

So, in Eiffel+ we shall use the term usecase both to refer to usecases, as well as, interaction diagrams. Thus, in Eiffel+ we shall use usecases for: specifying the system's requisites or functionality, specifying the agents external to the system (actors), and describing in detail the system's functionality: objects, methods and flow.

Eiffel+ uses usecases as a tool for verification and testing. Eiffel+'s compiler makes sure that all the objects involved in a usecase actually exist and that for each one of them the methods required to execute the usecase in question have been defined. On the other hand, the compiler also checks the particular flow in the usecase, and so it checks that the sequence or order in which the methods are called is correct.

Finally Eiffel+ will be able to tell us which parts of our system have not been used in any usecase, that is remain untested.

Relations between usecases: use and extension

Eiffel+ supports both use and extension relations between usecases as defined by Jacobson [(Ja92]).

In Eiffel+ we can specify the extensions to the usecases by making use of the use relation described earlier plus a conditional expression by which if the probe condition is verified the extending usecase (the usecase that extends) is executed.

Parameterized and Inherited Usecases

As was the case with relations, classes and clusters we may have parameterized usecases and inheritance between usecases. Either technique allows reusing usecases defined previously adapting them to new needs.

Usecases are descriptions of the functionality of a system or application. As [Ru91] explains, we may have prototypical architectures which describe characteristic designs. These architectures may be such as reactive, batch, real-time, etc. Each one of these architectures allows the user to access the system in certain ways. We can therefore typify the main interactions of the user with the system for each of these pre-defined architectural designs.

An example of this situation is the dialog that is established between user and system in maintenance programs, such as a customer or product maintenance program. We can, therefore, describe a parameterized usecase which describes the basic dialog that is established when creating, modifying or deleting application-domain objects from a master file.

Our description of use-cases in Eiffel+ is:

```
USECASE: USEcase identifier is
                actor LIST_OF_ACTORS
                [class LIST_OF_CLASSES]
                [cluster LIST_OF_CLUSTERS]
                [usecase LIST_OF_USECASES]
         do
           ACTOR_IDENTIFIER : STRING_ACTION
                 USECASE_BODY
           SYSTEM_REACTION_STRING : "<"ACTOR_IDENTIFIER">"
         end

USECASE_BODY: LIST_OF_ACTIONS
LIST_OF_ACTIONS: {ACTION, ....}
ACTION: REFERENCE ROUTINE_BODY end
      | usecase_identifier
```

The last type refers other usecases. This allows us to aggregate usecases into larger entities or to explode use cases to smaller ones.

```
REFERENCE: class_identifier.method_identifier
         | cluster_identifier.method_identifier
```

An interesting consequence has to do with user manuals. Since user manuals typically describe the way to use the system and this procedure is what the usecases describe, if we typify the usecases we will also be able to typify the user manuals. That is, we will be able to have skeleton manuals that correspond to parameterized usecases that will easily be adapted to each concrete application.

Inheritance and parametritzation has not been included for simplicity.

2.5 Specifying and documenting

Specification

This last section is dedicated to the specification of methods in the earlier stages of the software process. Due to the fact that the most difficult and tedious part of any software system is the development of the methods it is the most prone to be developed incrementally.

In the first place we must distinguish between four different situations in which we can have a method: implemented, deferred, pending, specified. The first two are supported by Eiffel. The last two will be introduced in Eiffel+.

An analyst may be aware of the need of a method but he may have to defer it's specification for further on. In a similar way as we may declare a method deferred we may declared a method **pending.** This notation has been used previously in VDM++ ([Lh94]) and has the advantage of allowing us to compile the unfinished class and execute it. The compiler will generate an exception if a pending method is accessed during run time.

Instead of leaving a method pending and undefined we have the alternative of specifying (not coding) the method. There are three different ways of specifying a method instead of coding: textually as comments, pseudo-code or with formal specifications.

We must discard the first way for it's ambiguities and imprecisions. Natural language descriptions are the source of the majority of communication errors.

Pseudo-code may be a valid alternative in many cases and it is used in many places. Sometimes it is called "structured english" and it is supported by many CASE tools, being the most frequent way analysts communicate programmers what a process must do. We propose using incomplete code. By this we mean that the analyst my describe the method using the complete instruction repertoire that Eiffel has. However he may leave whatever parts he wants unfinished if he adds a special comment such as <<initialize local variables>>

This way of specifying a routine is flexible because the level of precision is left open to the analyst and eliminates repeating descriptions between analyst and programmer. The programmer will only have to code the unfinished parts in a incremental form. By this we mean, as in the pending option described previously, that a class with a method incomplete may be compiled and executed. Only in the case of trying to access an incomplete branch will an exception arise informing the programmer which incomplete code is being tried to execute.

The last alternative is the formal specification. Eiffel has been a leader in the introduction of formal methods into a procedural language with its assertions. We propose extending Eiffel with three more clauses adopted from specification languages. The first one refers to the specification in each method of what attributes may be modified. This mat be done using the clause **modify** list of variables.

Eiffel+ will ensure that the only attribute that may be modified is balance. In the case of trying to modify any other attribute an exception will arise.

The second clause we propose to add is in the parameter definition of the routines. The parameters may be qualified as IN, OUT or INOUT as is done in other languages such as ADA.

The Eiffel+ compiler will ensure that IN parameters are non-void on input and not modified on finishing, evaluated with deep equality, OUT parameters are non-void on output and INOUT are non-void both on input and output.

Finally even though Eiffel does not have quantifiers for the assertions, it does allow the use of references to routines and therefore they may be simulated easily ,specially if we use predefined library iterators with deferred methods. Although this is true, it is also true that it is difficult to use functions because we are forced to leave the point were we want to reference the function and define that function elsewhere and then return to where we where previously. Another problem we have with this solution is that if we add a function to a class to be able to use a quantifier this function will be inherited in subclasses and they should have no access to it because it is not really a method of the class.

For all these reasons we propose the use of what we call "manifest functions". What we mean by this is that we will be able to instantiate directly the deferred class manifesting the deferred method. For example:

```
ensure
   !!ArrayIterator[Floodgate].make(floodgates).iteration
   where operation is
   do
      Result := not (floodgates.item(i).opened)
   end
```

Where operation is a deferred routine accessed from iteration.

The templates

Eiffel+ is documentary extensible. What we mean by this is that each computing center can define the documentation they want for each of the most import entities of the language. For this the analyst will have to define the structure of each template and compile them with a special utility. From then on the Eiffel+ compiler will expect all programs conform to the information required in the templates.

They will be able to define six templates: cluster, class, relation, state, routine and attribute. To describe the template we will use extended BNF with a few keywords for terminal symbols. Following we have an example used in chapter 1.2.3:

```
template routine:
        "Description:" text
        "Visibility:" (Private|Public|Protected")
        "Class:" (Yes|No)
        "Restrictions:" text
end
```

A scripting language will be defined to allow access to the Eiffel+ source to the analyst and to output it in whatever format he may need.

3. Comparison with related work

There are only two other languages proposed for activities of analysis and design. The first is ODL (Object Design Language) has been created by de Champeaux and is described in [Ch93]. De Champeaux suggests the use of a graphical notation (OAN, Object Analysis Notation) for analysis and a language in-between (between analysis and implementation) for design. The language is not executable and although the translation of ODL into such programming languages as C++ or Eiffel is also relatively simple it is not automatic. ODL cannot support any of the features Eiffel+ does, such as clusters, relations, state transitions or usecases.

The second language, OODSL has been created by Firesmith and is described in [Fi93]. OOSDL is defined as a strongly-typed, quasi-formal, textual, object-oriented

specification and design language. It is based on ADA ([Aj83]) and is influenced by Eiffel ([Me88]) and DRAGOON ([At91]).

It is not executable and it is an extensive language, with nearly one hundred reserved words and, like the earlier-mentioned ODL, is totally connected with its own graphical analysis notation, which, by way of reminder, includes 15 different diagrams, more than twice as many as any other and by no ways generalizable to other methodologies.

4. Conclusions

By way of conclusion, we want to point out that, if we want to ensure that object orientation provides a seamless transition, then, there is a natural need for the language to be the same at each stage, that is, there is a need for a software development language rather than a software programming language.

Secondly, the dividing line between what is programming and what is not is very thin and subjective, highly dependent on the language constructor.

Thirdly, we are often required to specify certain things at analysis and/or design levels, which, if we do not work with a full life-cycle language, makes it necessary to redefine or describe anew at the design or programming level, thus doubling the task and increasing the chances of introducing mistakes.

Fourthly, it is advisable for each activity to be described in the most natural way possible and that, afterwards, the compiler takes on the task of re-locating the descriptions in order to obtain an executable program.

Next, everything that is documented should be reused, if possible, for checking or validating the correctness of what we are building. Moreover, we must be able to cross-check the different views we have of a single system in order to ensure the consistency between them.

Finally, if all this, which is to a greater or lesser extent supported by object-oriented methodologies, is incorporated into one single language, then, we will

have accomplish the interoperability between the methods and the CASE tools that back up these methods.

Bibliography

Aj83 Ada Joint Program Office (1983)
Reference Manual for the Ada Programming Language; United States Department of Defense

At91 Atkinson C. (1991)
Object-Oriented Reuse, Concurrency and Distribution; Addison Wesley

Bo91 Booch G. (1991)
Object-Oriented Design with Applications; Benjamin-Cummings

Bo93 Booch, G (1993)
Object-oriented Analysis and Design with Applications, 2nd Ed., Benjamin Cummings

BoRu96 Booch, Rumbaugh (1996)
The Unified Method, Ver. 0.8, Rational

Ch93 de Champeaux D, Lea D., Faure P. (1993)
Object-Oriented System Development; Addison-Wesley

Ck94 Cook S, Daniels J (1994)
Designing Object Systems. Object-Oriented Modelling with Syntropy, Prentice Hall

Cl94 Coleman D. et al. (1994)
Object-Oriented Development: The Fusion Method, Prentice-Hall

Co91a Coad P., Yourdon E. (1991)
Object-Oriented Analysis 2nd Edition; Prentice Hall

Co91b Coad P., Yourdon E. (1991)
Object-Oriented Design; Prentice Hall

Co93 Coad P., Nicola J. (1993)
Object-Oriented Programming; Prentice Hall

Do95 Dodani, Velazquez (1995)
Supporting Interactions as First-Class Citizens in OO Methodologies, ROAD Vol. 2 No. 4, Nov.-Dec. 1995

Fi93 Firesmith D. (1993)
Object-Oriented Requirements Analysis and Logical Design, A Software Enginerring Approach; Wiley

Ga94 Gamma E., Helm R., Johnson R., Vlissides J. (1994)
Design Patterns: Elements of Object-Oriented Software Architecture; Addison-Wesley

Gr94b Graham I. (1994)
Migrating to Object Technology; Addison-Wesley

Ha87 Harel D. (1987)
Statecharts: avisual formalism for complex systems, Science of Computer Programming 8:231-274

He94a Henderson-Sellers B., Edwards J. (1994)
Booktwo of Object-Oriented Knowledge: The Working Object; Prentice-Hall

Ja92 Jacobson I. et al. (1992)
Object-Oriented Software Engineering: A Use Case Driven Approach, 4th Ed. Addison-Wesley

Kr94 Kristen G. (1994)
Object Orientation, The Kiss Method, From Information Architecture to Information System; Addison-Wesley

Lh94 Lano K., Haughton H. (1994)
Object-Oriented Specification Case Studies; Prentice Hall
Ma92 Martin J, Odell J. (1992),
Object-Oriented Analysis and Design, Prentice Hall
Ma93 Martin J (1993),
Principles of Object-Oriented Analysis and Design, Prentice-Hall
Ma94 Martin J, Odell J. (1994),
Object-Oriented Methods: A Foundation, Prentice Hall
Me88 Meyer B (1988),
Object-Oriented Software Construction, Prentice Hall
Me92a Meyer B. (1992)
Eiffel: The Language"; Prentice-Hall
Ne94 Nerson JM, Walden K. (1994)
Seamless Object-Oriented Software Architecture; Prentice Hall
Od94a Odell J. (1994)
Six diferent kinds of composition; Journal of Object-Oriented Programming.
Vol. 5 No. 8
Pe94b Peralta, A, (1994)
Making the Transition from ADTs to Objects in Undergraduate Software
Engineering: a CASE-based Approach, 1994 Joint Modular Language
Conference
Ro92 Robinson P. (1992)
Hierarchical Object-Oriented Design; Prentice Hall
Ru91 Rumbaugh J. et al. (1991)
Object-Oriented Modeling and Design, Prentice-Hall
Sh88 Shlaer S, Mellor S.J. (1988),
Object-Oriented Systems Analysis: Modelling the World in Data, Yourdon
Press
Sh91 Shlaer S, Mellor S.J. (1991),
Object LIfecycles: Modelling the World in States, Yourdon Press
So95 Sourrouville, Lecorude (1995)
A Dynamic Model Extension to Integrate State Inheritance and Objects, ROAD
Vol. 2 No. 2, July-August 1995
Wi90 Wirfs-Brock RJ, Wilkerson B., Wiener L. (1990)
Designing Object-Oriented Software; Prentice Hall

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Research Reports – 1996

LSI-96-1-R "(Pure) Logic out of Probability", Ton Sales.

LSI-96-2-R "Automatic Generation of Multiresolution Boundary Representations", C. Andújar, D. Ayala, P. Brunet, R. Joan-Arinyo, and J. Solé.

LSI-96-3-R "A Frame-Dependent Oracle for Linear Hierarchical Radiosity: A Step towards Frame-to-Frame Coherent Radiosity", Ignacio Martin, Dani Tost, and Xavier Pueyo.

LSI-96-4-R "Skip-Trees, an Alternative Data Structure to Skip-Lists in a Concurrent Approach", Xavier Messeguer.

LSI-96-5-R "Change of Belief in SKL Model Frames (Automatization Based on Analytic Tableaux)", Matías Alvarado and Gustavo Núñez.

LSI-96-6-R "Compressibility of Infinite Binary Sequences", José L. Balcázar, Ricard Gavaldà, and Montserrat Hermo.

LSI-96-7-R "A Proposal for Word Sense Disambiguation using Conceptual Distance", Eneko Agirre and German Rigau.

LSI-96-8-R "Word Sense Disambiguation Using Conceptual Density", Eneko Agirre and German Rigau.

LSI-96-9-R "Towards Learning a Constraint Grammar from Annotated Corpora Using Decision Trees", Lluis Màrquez and Horacio Rodríguez.

LSI-96-10-R "POS Tagging Using Relaxation Labelling", Lluís Padró..

LSI-96-11-R "Hybrid Techniques for Training HMM Part-of-Speech Taggers", Ted Briscoe, Greg Grefenstette, Lluís Padró, and Iskander Serail.

LSI-96-12-R "Using Bidirectional Chart Parsing for Corpus Analysis", A. Ageno and H. Rodríguez.

LSI-96-13-R "Limited Logical Belief Analysis", Antonio Moreno.

LSI-96-14-R "Logic as General Rationality: A Survey", Ton Sales.

LSI-96-15-R "A Syntactic Characterization of Bounded-Rank Decision Trees in Terms of Decision Lists", Nicola Galesi.

LSI-96-16-R "Algebraic Transformation of Unary Partial Algebras I: Double-Pushout Approach", P. Burmeister, F. Rosselló, J. Torrens, and G. Valiente.

LSI-96-17-R "Rewriting in Categories of Spans", Miquel Monserrat, Francesc Rosselló, Joan Torrens, and Gabriel Valiente.

LSI-96-18-R "Strong Law for the Depth of Circuits", Tatsuie Tsukiji and Fatos Xhafa.

LSI-96-19-R "Learning Causal Networks from Data", Ramon Sangüesa i Solé.

LSI-96-20-R "Boundary Generation from Voxel-based Volume Representations", R. Joan-Arinyo and J. Solé.

LSI-96-21-R "Exact Learning of Subclasses of CDNF Formulas with Membership Queries", Carlos Domingo.

LSI-96-22-R "Modeling the Thermal Behavior of Biosphere 2 in a Non-Controlled Environment Using Bond Graphs", Angela Nebot, François E. Cellier, and Francisco Mugica.

LSI-96-23-R "Obtaining Synchronization-Free Code with Maximum Parallelism", Ricard Gavaldá, Eduard Ayguadé, and Jordi Torres.

LSI-96-24-R "Memoisation of Categorial Proof Nets: Parallelism in Categorial Processing", Glyn Morrill.

LSI-96-25-R "Decision Trees Have Approximate Fingerprints", Víctor Lavín and Vijay Raghavan.

LSI-96-26-R "Visible Semantics: An Algebraic Semantics for Automatic Verification of Algorithms", Vicent-Ramon Palasí Lallana.

LSI-96-27-R "Massively Parallel and Distributed Dictionaries on AVL and Brother Trees", Joaquim Gabarró and Xavier Messeguer.

LSI-96-28-R "A Maple package for semidefinite programming", Fatos Xhafa and Gonzalo Navarro.

LSI-96-29-R "Bounding the expected length of longest common subsequences and forests", Ricardo A. Baeza-Yates, Ricard Gavaldà, and Gonzalo Navarro.

LSI-96-30-R "Parallel Computation: Models and Complexity Issues", Raymond Greenlaw and H. James Hoover.

LSI-96-31-R "ParaDict, a Data Parallel Library for Dictionaries (Extended Abstract)", Joaquim Gabarró and Jordi Petit i Silvestre.

LSI-96-32-R "Neural Networks as Pattern Recognition Systems", Lourdes Calderón.

LSI-96-33-R "Semàntica externa: una variant interessant de la semàntica de comportament" (written in Catalan), Vicent-Ramon Palasí Lallana.

LSI-96-34-R "Automatic verification of programs: algorithm ALICE", V.R. Palasí Lallana.

LSI-96-35-R "Multiresolution Approximation of Polyhedral Solids", D. Ayala, P. Brunet, R. Joan-Arinyo, I. Navazo.

LSI-96-36-R "Algebraic Transformation of Unary Partial Algebras II: Single-Pushout Approach", P. Burmeister, M. Montserrat, F. Rosselló, and G. Valiente.

LSI-96-37-R "Probabilistic Conditional Independence: A Similarity-Based Measure and its Application to Causal Network Learning", Ramon Sangüesa Solé, Joan Cabós Fabregat, and Ulises Cortés García.

LSI-96-38-R "Analysing the Process of Enforcing Integrity Constraints", Enric Mayol and Ernest Teniente.

LSI-96-39-R "Reducció de l'equivalència inicial visible a teoremes inductius" (written in Catalan), Vicent-Ramon Palasí Lallana.

LSI-96-40-R "A Compendium of Problems Complete for Symmetric Logarithmic Space", Carme Àlvarez and Raymond Greenlaw.

LSI-96-41-R "Semàntica algebraica del llenguatge AL: l'algorisme α" (written in Catalan), V.R. Palasí Lallana.

LSI-96-42-R "Partial Occam's Razor and its Applications", Carlos Domingo, Tatsuie Tsujiki, and Osamu Watanabe.

LSI-96-43-R "Transparent Distributed Problem Resolution in the MAKILA Multi-Agent System", Karmelo Urzelai.

LSI-96-44-R "The Intensional Events Method for Consistent View Updating", Dolors Costal, Ernest Teniente, and Toni Urpí.

LSI-96-45-R "Extending Eiffel as a Full Life-cycle Language", Alonso J. Peralta and Joan Serras.

Hardcopies of reports can be ordered from:

Nuria Sánchez
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo, 5
08028 Barcelona, Spain
secrelsi@lsi.upc.es

See also the Department WWW pages, http://www-lsi.upc.es/www/