ISTITUTO NAZIONALE DI FISICA NUCLEARE

Sezione di Milano

INFN/TC-04/05
25 Marzo 2004

STUDY OF A PURE OBJECT PROGRAMMING LANGUAGE FOR RELIABLE SYSTEM DEVELOPMENT

Matteo Sassi

INFN, Sezione di Milano and Dipartimento di Fisica , Università degli Studi di Milano - via Celoria 16, I-20133 Milano, Italy

Abstract

In nuclear physics the use of object oriented programming languages is continuously growing. Object programming is used in the development of data analysis frameworks such as ROOT from CERN and the SMI++ one used by DELPHI experiment. The object programming languages allow a rapid and flexible development of the data analysis software and acquisition control code, and a reduction of development and maintenance costs, because each system is viewed as a collection of objects. However the use of object oriented programming in the development of dependable systems introduces the reliability problem. To solve this problem a Pure Object Language Core (POLC) has been studied, based on the formal theory of the objects. In the present work this study is discussed. An example of application based on an optimum digital filter for gamma high resolution spectroscopy is presented.

PACS.: 29.50.+v, 89.20.Ff, 07.05.Bx, 07.05.-t

Published by ***SIS–Pubblicazioni***
Laboratori Nazionali di Frascati

1 Introduction

In nuclear physics the use of object oriented programming languages is continuously growing. Examples of object programming use are the development of data analysis frameworks such as ROOT (1), the SMI++ framework (2) in DELPHI experiment and the large use of object programming in the BABAR experiment (3). Anyhow the whole scientific area is interested to the large use of object oriented programming languages, like C++ and JAVA.

The programming languages allow a rapid and flexible development of the data analysis software and acquisition control code. In fact in the object oriented programming is possible to reduce the development and the maintenance costs by reusing library of software components developed for previous experiments and easily upgradable to fulfil new demands.

In the field of dependable systems development the use of object oriented programming introduces some drawbacks. In software engineering context the object oriented programming, in particular the use of commercial languages like C++ and JAVA, is in contrast with the demand of reliability of dependable systems (4). In order to develop reliable software systems it will be necessary to follow strict constraints. The space agencies provide more detailed indications on the development of reliable software. The NASA (5) asserts that the use of object oriented programming languages reduces design errors prior to coding. Anyhow the use of commercial object oriented programming languages is not suitable because not strictly supported by a formal theory.

Consequently the development of safety critical systems is in contrast with the request to decrease system planning and development costs. The indications of NASA (5), ESA (6) and ASI (7) are to seek a “safe subset” of commercial object oriented language.

In the present work a Pure Object Language Core (POLC) is introduced. It is a set of structures that defines a pure object programming language based on the λ-Calculus formal theory (8): the object calculus.

2 Software Architecture and Object Programming

An object oriented programming language is based on the cooperation of a definite set of elements called *objects*. In Fig. 1 a typical object system consisting of a set of objects that are interconnected by a specific architecture is shown. The connecting solid lines represent the control flow and the data flow of the system.

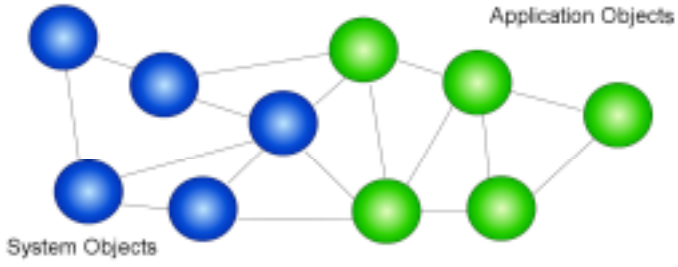

Fig. 1: Example of object system.

The object programming languages establish a specific correspondence between the system software simulation and the real system to make the modelling and planning of the complex systems more immediate and intuitive.

In the imperative programming language the basic element is the single instruction. In consequence the abstraction distance between the software model and the real system is high. In imperative programming the software model is composed by modules. Each module contains instructions sequences and share data areas. On the contrary the object programming methodology is based on the finding out of a software model, on its analysis and design. This model consists of a set of software components, the objects, and their relationships. This methodology simplifies the implementation of a software architecture (4) starting from a high definition level and refines it to a lower description level. In this way a more elevated level of system modelling is reached than with the standard program design. This causes a simplification of the system analysis and design, particularly in great and complex systems.

Software architectures are commonly used and defined in software engineering. In physical experiments and space science the design of complex, robust and reliable programs leads to a top-down approach. This approach is based on the subdivision of the whole design problem in simple ones, according to a high-level architecture model, and on the refining of requirements in the lowest level. Software architectures are developed and evaluated at each abstraction level. They have some typical properties, such as computation locality, information hiding, error control, diagnosability, upgradability. Moreover it is necessary to assure the quality of the software process by fitting a proper architecture and design method.

The fast system development, the system maintainability and reliability must be ensured by the software development models. However, the development of a complex system depends on the system size and functionalities, and on the number of controlled items. In complex systems the design must be distributed to different teams. The architectural choice allows to subdivide the project among teams without loss of assurance.

An object oriented architecture is well suited to support complex system designs because each object is well characterized. Moreover the object oriented approach separates each object activity by providing an interface between the object service and the rest of the system. The aim of objects programming is the development of a distributed system of elements that cooperate in the program execution. The design of distributed software architectures meets the request of resources sharing, concurrency, scalability, fault tolerance, and transparency.

The use of objects makes more flexible the software design. The object languages allow the easy substitution, implementation or addition of software components that can consist of both simple objects and subsystems of more objects. This is possible by using already existing components libraries and subsystems, and consequently allows a rapid development and implementation of the system while keeping the abstraction level and the intuitiveness of the object languages.

2.1 The abstraction level in the programming

The object design uses the object abstraction from the beginning of the system development. The abstraction concept is fundamental in programming languages and the modelling of any system is based on it. The abstraction mechanism allows the representation of the target system at a detailed level different from the final one. This emphasizes the interested details and neglects the unessential ones at this description level.

Different types of abstraction exist in a programming language, such as: the procedural abstraction, the data abstraction and the iterative abstraction (9). The *procedural abstraction* is based on the division of the system operations in subroutines called during the program execution. The procedure is the basic element of this abstraction; it allows to transform input data into output data. This abstraction conceives the program as a procedure containing a list of subroutines. The algorithms defined in this way are far from the system model. The *iterative abstraction* is based on the repetition of operations sequences in cycles defined by the programmer. The *data abstraction* is based on the possibility to associate an exact structure and an exact set of operations to every data in order to characterize it.

These abstractions levels coexist in any programming language and furnish the context to develop the system. Typically, these abstraction levels disappear while the program is written because the language elements are instructions. This is the reason why the distance among the abstraction design level and the flow of the instructions is greater in the procedural languages than the object oriented programming languages.

2.2 Object programming languages

The object programming language derives from the type theories (10), in particular from two theoretical frameworks: the abstract data types and the polymorphous types. These theories represent a formal context for the correctness proof of a program. The formal proof consists of a validation of the program in this abstraction level.

In the abstract data types theory (11) a formal treatment of data types is provided. An abstract data type is composed by data with the same structure and the same set of operations. Examples of different abstract data types are: the integer, real and complex types together with their structure and the definite operators for manipulating them.

The polymorphous types theory is based on the concept of polymorphism and on the classification of the data types in sets of similar structures (12). This theory introduces the concept of subtype deriving a new data structure from an already existing one. This introduces the object oriented programming mechanism, known as inheritance.

The object is the basic component of the software model of these programming languages. The object is characterized by the data structure and the methods for accessing and processing data. The concept of object is an extension of abstract data types (12). The object provides the capability to perform actions and computation on its data area.

It is possible to make a similitude between the object programming and the procedural programming, being the concept of object more specific than a simple extension of the imperative programming. In fact the object abstracts all the actions permitted from the data

type. This is possible because the object, in a real system as in a software system, is identified by a group of quantities and specific modalities of use.

An object has a structure mainly composed by: *Object Name*, *Data Area*, *and Method List*. The object evolution is characterized by a set of properties: the method invocation and the method lookup, the encapsulation of data area (public, private), the inheritance and the subsumption (8). In each object programming language, the object structure and its properties are present at different levels, each language implementing these characteristics in different ways.

The classification of object programming languages is based on two categories: the first one defines pure object programming languages or object oriented programming languages and the second one defines class based languages or object based languages (8). In the Tab. 1 the classification of some commercial and academic programming languages is shown.

Tab. 1: Commercial and academic programming languages classification.

	Object Based	**Class Based**
Pure Object	*Self (13), Cecil (14)*	*Blue (15)*
Object Oriented	-	*C++, Java*

In *pure object languages* all is an object; no variables or procedures can be used by the programmer. Each element of the software is modelled by objects, by their data areas, and by their methods. In the *object oriented programming languages* the concept of object is still present, moreover there are concepts of procedures and variables descending from procedural programming (16).

The *class based languages* use the classes as a static description of objects. Each object is derived by a class. This class represents the abstract model of the object. The process of object creation is called instantiation. The instantiation transforms a static element, the class, in a dynamic element, the object. The object instantiation is a primitive concept of class based languages. On the contrary the *object based languages* are more direct because the basic elements of the programming are the objects. The programmer defines the criterion for the object creation. The object creation is similar to instantiation.

The object dynamical evolution during the execution is the main problem in the Object Oriented Programming Languages. In particular, the model of method lookup is not evident during method invocation in case of derived classes. The aim of this work is to define a dynamical evolution model that consents to determine the program execution. Moreover the definition of a programming language must be inserted in a formal context.

The λ-Calculus constitutes the formal context to study the object programming languages and provides the proof of its universality. The λ-Calculus is presented and defined in (8), while for an informal description see (17). A Calculus defines the rules to build its elements in a formal system and the rules to manipulate such elements. These rules characterize the primitive elements of the Calculus and their properties.

The □-Calculus is the untyped object calculus. It provides the properties and the base operations to define and to use the objects. The primitives of this Calculus are the objects. A direct semantic is defined in the Calculus, it fixes the rule set on the interpretation and on the computation. The objects are the only computational structure permitted in the Calculus, thus supplying an advantageous abstraction by conforming each model element to the object concept. The □-Calculus introduces the least number of primitives and allows the deriving of the more complex constructions. The definition of the object is minimal, essential and self consistent.

A □-Calculus object is composed by a simple collection of members. These are methods and fields. The operations permitted in the objects are the *selection* and the *updating* of a member.

3 POLC: a Pure Object Language Core

POLC is an object language that incorporates the primitive concepts of the □-Calculus and the functionalities that allow the calculus execution.

The □-Calculus provides the primitive formal semantics of a pure object language; the dynamic of the operations of method selection and field updating are implicit in the semantics of the □-Calculus. The *updating* allows to modify the fields. The *selection* of the method starts the execution of the method body. These two actions have to be integrated in the structure and in the operational semantics of the object.

Tab. 2: POLC Language Assumptions.

1	all is an object.
2	the communication happens through the messages exchange (message passing).
3	the only two operations permitted on an object are the selection and the updating.
4	each object has an associated status that represents the actual evolution point.

The language is based on the assumptions shown in Tab. 2. Each language element is an object. This implies that each software component has the same abstraction level, avoiding the use of different abstraction levels during the modelling. Consequently the software design and the software development of the system are simplified. Furthermore the system is organized into objects and the object is the only modelling element of the system.

The language elements communicate through an exchange of messages. In this way the modalities of the method-call and the data-exchange among the objects reduce to a message dispatch. Moreover the access to shared memory is eliminated because each object manages the field access providing the information hiding. The communication among objects is structured by a protocol. The advantage is to uniform the technique of information exchange and to simplify the debug of the software system.

The only permitted operations on each object are the *selection* and the *updating*. This drastic reduction of the accesses typology to the software component allows the testability during the system development and the debug without loss of generality.

Every object has an exact status that represents the computational evolution in the system, allowing the monitoring of the computation evolution during the execution. The language also provides an operational semantics based on status evolution. Such semantics is contained in the state transition function discussed later.

This allows a more simple maintenance of the system in the case of software faults, errors or new version releasing. The maintenance activity only requires to find the interested group of objects and patch them in accordance with the demands. This is possible because the POLC object defines specified utilization methods of object, using the languages assumptions.

3.1 Abstract object definition

A POLC object is represented by the structure shown in Fig. 2. Each element of this structure has a specific functionality in the object. The *ObjId* (object identifier) contains the information needed to distinguish the object in the software system. The object characterization is performed by the *member list*, containing the method definitions and the data structure. The *message manager* deals with messages coming from and sent toward the other objects of the software system. This element deals with the management of the object communication. The *state* and the *evolution model* embody the operational semantics of the language to determine the correct system operation.

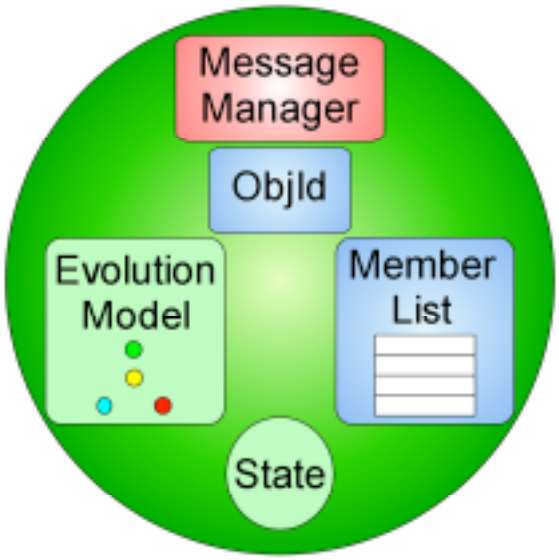

Fig 2: POLC Object Structure, different sections are in the same colour.

Therefore the POLC object is composed by three sections: a Static section (ObjId and Member List), a Communication section (Message Manager) and a Dynamic section (Evolution Model and State). This makes POLC different from the other programming languages where it is typically present only the syntactic definition of the software element.

The Evolution Model and the Message Manager have to be considered two primitive characteristics of every POLC object.

3.2 Language Syntax

The POLC syntax is simple and essential, allowing a rapid code development and an efficient debug of the system. POLC is in the development phase and its syntax is in evolution to allow a higher abstraction level in the software system modelling. It is possible to declare an object using the essential syntax of Tab. 3. The syntax allows the declaration of every computation element.

Tab. 3: Example of Essential POLC Syntax.

Object Declaration	Method Declaration
OBJECT ObjectName MemberNumber *MemberLabel1 Type Body* … … *MemberLabelN Type Body* *ENDOBJECT;*	*METHOD ObjectName :: MethodName (MessageIn)* *// instructions* *return (MessageOut)* *ENDMETHOD;*
Message	
(ObjSd) – the sender identifier; *(ObjRd) – the receiver identifier;* *(MemberName) – the name of the interested member;* *(Parameter) – the parameters of the message;* *(Selection or Update) – the indicator of the in demand operation.*	

The object declaration starts with the keyword “OBJECT” and stop with “ENDOBJECT”. The “ObjectName” represents the univocal identifier of the object; “MemberNumber” indicates the number of members contained in the object. Each line included between “OBJECT” and “ENDOBJECT” indicates the declaration of a single member. “MemberLabel” identifies the name of the member; “Type” indicates if the member is a field or a method; “Body” may contain the method code or the value of the field.

Tab. 4: POLC Method Rules.

1	Every method is able produce one or more output messages.
2	The method can access in reading the value of all the members of the object.
3	The method is unbreakable and the call to other methods is permitted only through output messages.
4	The method is able to update the value of one or more members only through output messages.
5	The method is able to use library functions incorporated in the method body.

The Method declaration is a thread of instructions using the POLC construction rules of Tab. 4. These rules make the POLC method a basic unit of execution. The method rules make the method as an unbreakable block of instructions entirely independent from the other computation elements. Each object method can access to its object fields and the direct access to other object fields is forbidden. The access to other object fields must be provided by the programmer defining a proper method.

3.3 Operational Semantics and Dynamical Evolution

The Π-Calculus provides the operations of *Updating* and *Selection* of a member. In POLC, the request of operation execution happens through a message. The message is directly dispatched to the object that contains the interested member. The updating operation modifies the body value of the member. The selection executes the instructions contained in the method body and produces one or more output messages that are the output of the computation.

The object evolution dynamic consists of the computation evolution and of the communication among the objects. POLC delegates part of the computation activity to communication. The status of the resources in execution and the object availability are provided by the evolution dynamic. The limitation of execution resources and the complexity of communication network are the main cause of the dynamical evolution problem. The language does not specify the management of this activity. The communication based on messages causes a bigger system complexity but advantages the information exchange observability that it is typically hidden in the memory access.

The status shows the actual object operational condition. This is determined by the computation evolution. Moreover the status indicates the performed object action or the object availability to perform other operations. The transition function is modelled by a finite state machine. This function and the allowed state for an object are shown in Fig. 3. The transition function determines the object evolution across the characteristic status. It allows to establish if an element is available to perform the expected operation.

The state transition function is the formal model of the language operational semantics. The advantage of this approach is to provide the evolution model of the computational element. This allows to make system behaviour forecasts, to monitor also the system evolution during the execution of computation, and to estimate the bottleneck of computation and communication, the real execution resources occupation, and the objects in error state.

The state transition function makes the POLC object an element of asynchronous computation. This allows the automatic extraction of the intrinsic parallelism in the software system. The execution requests are stored into the message manager and executed by available resources at a proper time. The computation can start without any control system or synchronization. This partially simplifies the programmer job. Moreover an asynchronous system allows the development of a synchronous one satisfying the request for a central control system through the use of handshaking techniques or control models. These control models must be provided by the system programmers.

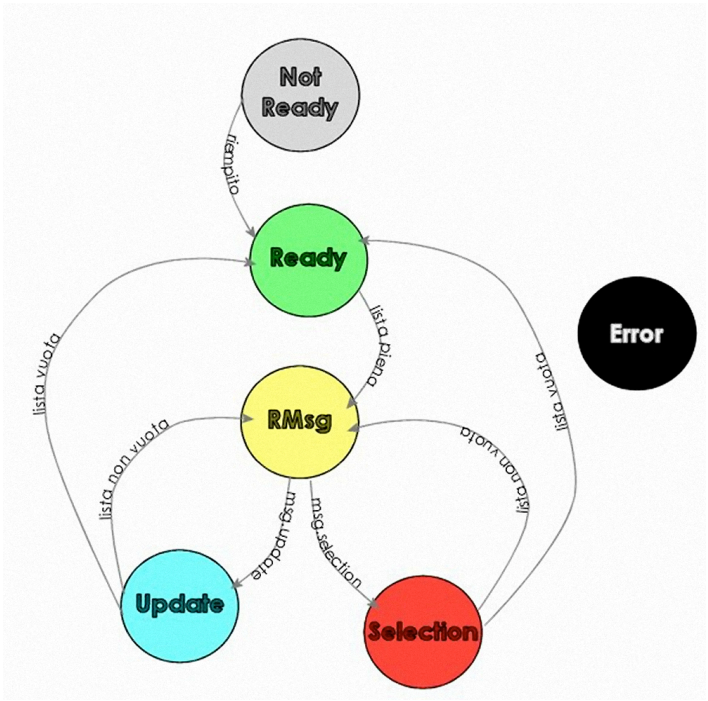

Fig. 3: POLC Transition Function definite by a Finite State Machine.

The communication among the objects is provided by the dispatch of messages. The message structure is: {*ObjSd*; *ObjRd*; *MemberName*; *Parameter*; *Selection* or *Updating Flag*} as shown in Tab. 3. This structure contains all the information needed to execute an operation.

One or more messages are produced in output of the method selection. When an object receives a message, inserts the message in the input message list, later the message manager solves the message list following the chronological arrival order.

4 Description of an Application

In this section an example of POLC application is provided. The application is a trapezoidal pulse shaper typically used as optimum digital filter for gamma high resolution spectroscopy (18). A typical hardware implementation of this filter is shown in Fig. 4.

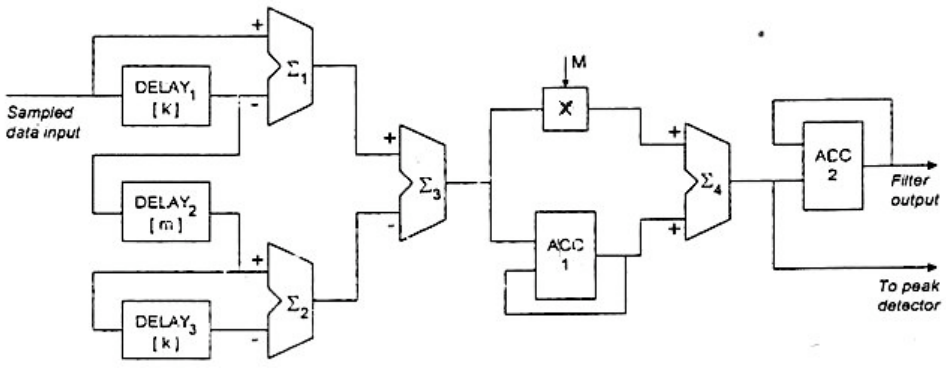

Fig. 4: Hardware block diagram of trapezoidal pulse shaper (18).

The usual imperative programming implementation of this filter is a digital recursive algorithm, defined by a set of numerical sequences as shown in Eq. 1, Eq. 2, and Eq. 3. The numerical sequence *s(n)* represents the output of the trapezoidal shaper at time *n*; *v(n)* is the input value at time *n*; *l* and *k* are two delay parameters and *M* is a multiplicative parameter. The algorithm is defined by a set of sequential instructions.

$$d^{k,l}(j) = v(j) - v(j-k) - v(j-l) + v(j-k-l) \quad (1)$$

$$p(n) = p(n-1) + d^{k,l}(n) \quad (2)$$

$$s(n) = s(n-1) + p(n) + M \cdot d^{k,l}(n) \quad (3)$$

On the contrary in the object programming, the software architecture of a program consists in a set of object properly interconnected. It is possible to use the hardware block diagram as model of software system. This model of the system consists in the defining of each object and its input and output connections. The software system is simply structured.

The program software architecture of the trapezoidal pulse shaper is composed by five object types that encapsulate the basic functionality of the system. This object types are:

- the *Delay* object, that furnishes the output value with a fixed delay;
- the *Adder* object, that makes the algebraic sum of two input values;
- the *Accumulator* object, that sums the input value and the last value;
- the *Multiplier* object, that multiplies the input value by a fixed parameter;
- the *FanOut* object, that subdivides the input value in two outputs.

The software architecture of the trapezoidal pulse shaper is shown in Fig. 5. It is composed by:

- three Delay objects (D1, D2, D3, D4, D5);
- four Adder objects (S1, S2, S3, S4);
- two Accumulator objects (A1, A2);
- four FanOut objects (F1, F2, F3, F4).
- one Multiplier object (M);

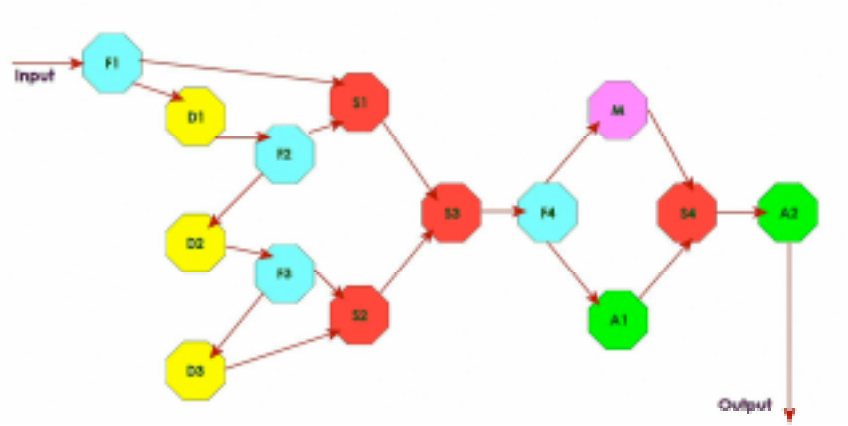

Fig. 5: Application software architecture of the trapezoidal pulse shaper.

Comparing Fig. 5 and Fig. 4 is possible to note that the software architecture is similar to the hardware block diagram. Each hardware functional block is modelled by an object type and the programmer can use the objects as hardware components.

4.1 Program Definition

The program is composed by fourteen objects derived from five basic object types. In the following the syntactical definition of each object type is discussed. Each object type is composed by a set of fields and methods. Each method is associated to a specific method body that is written in C++. An example of a method body is discussed later.

Delay

The delay object is similar to a delay line. Each input value is stored in a buffer and concurrently the output value is sent to a destination object. The delay object is composed by ten fields: from C1 to C7 are used as a buffer vector; C8 is used as vector index of actual output values and C9 used as vector dimension. The field C10 contains the information needed for linking the delay object to other objects in the system. In this example the object is connected to the method M2 of OBJ1.

The method M1 reads the output value from the buffer vector (C1..C7) using the index C8, sends a message to C10 linked object with the output value, and to itself for the selection of M2 methods.

The method M2 increments the C8 value in the circular range from 1 to the value stored in C9, sends the C8 update message, and the update message to the buffer vector with the input value.

```
OBJECT OBJ2 12
    C1   F    0
    C2   F    0
    C3   F    0
    C4   F    0
    C5   F    0
    C6   F    0
    C7   F    0
    C8   F    1
    C9   F    7
    C10  F    OBJ1;M2
    M1   M    F5
    M2   M    F6
ENDOBJ;
```

Adder

The adder object is composed by the fields C1 and C2 used as registers, and by the C3 field that contains the information needed for linking the adder object to other objects in the system. The methods M1 and M2 set the fields C1 and C2 respectively; while the method

M3 performs the adder operation. The method M4 resets the adder object fields to the initial value.

The method M1 (M2) sets the field C1 (C2) with the value of the message parameter and checks the value of C2 (C1). If the value of C2 (C1) is not EMPTY a message of method M3 selection is sent.

The method M3 verifies that C1 and C2 are not EMPTY. In the following it performs the algebraic operation C1 - C2 and sends the obtained results to the C3 linked object. Moreover it sends the message for the selection of method M4.

The method M4 sets the value of C1 and C2 fields to EMPTY using two update messages.

```
OBJECT OBJ4 7
    C1   F    EMPTY
    C2   F    EMPTY
    C3   F    OBJ10;M1
    M1   M    F1
    M2   M    F2
    M3   M    F3
    M4   M    F4
ENDOBJ;
```

Accumulator

The accumulator object is composed by the C1 field used as actual accumulated value, and by the C2 field that contains the information needed for linking the accumulator object to other objects in the system. The method M1 performs the accumulator operation.

The method M1 sums the input value to the C1 value. The result is sent to C2 linked object and to itself for C1 updating.

```
OBJECT OBJ13 3
    C1   F    0
    C2   F    OBJ14;M1
    M1   M    F7
ENDOBJ;
```

FanOut

The FanOut object is a linking object. It is composed by two fields and one method. The fields C1 and C2 contain the information needed for linking the FanOut object to other objects in the system. The method M1 dispatches the input message to the C1 linked object and to the C2 linked object.

```
OBJECT OBJ6 3
    C1   F    OBJ8;M1
    C2   F    OBJ7;M1
    M1   M    F9
ENDOBJ;
```

Multiplier

The multiplier object is composed by two fields and one method. The field C1 contains the multiplicative parameter; the field C2 contains the information needed for linking the multiplier object to other objects in the system; the method M1 performs the multiplier operation.

The method M1 multiplies the input value and the C1 value. The result is sent to the C2 linked object.

```
OBJECT OBJ12 3
    C1   F    -3
    C2   F    OBJ14;M2
    M1   M    F8
ENDOBJ;
```

3.2 Method definition

A typical example of method body written in C++ is presented. The method body is F2 and is used by the adder object. The method updates the field C2 with the message parameter, and sends the message for the selection of method M3.

```
TMessageObjOut TMethodList::F2
          (const TMessageObj Message, const TAttribute * Attribute, const int AttrNum)
{       /* controlling if C2 value is EMPTY
                if yes  inserts the parameter value in to MINUS and sends the proper messages
                else    sends a BUSY message to the sender object */
TMessageObjOut MessageOut;
int i;
char MINUS[LATTRIBUTE];
for (i=0; i<AttrNum; i++) if (strcmp(Attribute[i].Label,"C2")==0)
strcpy(MINUS,Attribute[i].Body);
if (strcmp(MINUS,"EMPTY")==0)
        {       // updates message of C2 with the message.param
          strcpy(MessageOut.Message[0].ObjRd,Message.ObjRd);
          strcpy(MessageOut.Message[0].ObjSd,Message.ObjRd);
          strcpy(MessageOut.Message[0].AttrId,"C2");
          strcpy(MessageOut.Message[0].Param,Message.Param);
          MessageOut.Message[0].selection = false;
          // selection of M3 to start sum
          strcpy(MessageOut.Message[1].ObjRd,Message.ObjRd);
          strcpy(MessageOut.Message[1].ObjSd,Message.ObjRd);
          strcpy(MessageOut.Message[1].AttrId,"M3");
          strcpy(MessageOut.Message[1].Param,"");
          MessageOut.Message[1].selection = true;
        }
else
        {       // the object is BUSY. It does not execute the previous sum!
          strcpy(MessageOut.Message[0].ObjRd,Message.ObjSd);
          strcpy(MessageOut.Message[0].ObjSd,Message.ObjRd);
          strcpy(MessageOut.Message[0].AttrId,"BUSY");
          strcpy(MessageOut.Message[0].Param,"BUSY");
          MessageOut.Message[0].selection = false;
        }
return (MessageOut);
}
```

The method copies the C2 value in the local variable MINUS. Then it controls the C2 value. If it is EMPTY, two messages are generated. The first is the update of C2 with the right value, the second is the selection of M3. Else the adder object does not performed the previous sum and sends a BUSY message to the object who has request the execution of this method.

5 Summary

POLC is a pure object language suited to the development of dependable software systems. It is based on the □-Calculus formal theory and it merges the primitive concept of object and its semantics in a formal evolution model. At the moment POLC object is implemented under C++. POLC provides a simple object language to study and develop a software system.

Each POLC language element is an object that simplifies the software development. Furthermore the system is organized through objects and the object is the only modelling element of the system. This allows a more simple maintenance in case of faults, errors or updating. The communication among the language elements happens through the exchange of messages. This simplifies the modality of method-call and data-exchange among the objects by reducing them to a message dispatch. This language entirely eliminates the access to shared memory because each object manages the data area access providing the information hiding. The communication among system elements is structured by a protocol. The advantage is to uniform the technique of information exchange and to simplify the debug of the software system. POLC reduces the access typology of the software component allowing testability and debug during the system development. POLC objects have a status that represents the computational evolution in the system. This allows to monitor the evolution of the computation during the execution. The state transition function is modelled by a finite state machine and it is the formal model of the language operational semantics.

The POLC expressivity and computational power have been empirically evaluated using an object model written in C++ and running under a C++ simulator. An example of application has been discussed.

An Object Abstract Machine (ObAM) based on POLC has been developed (19). The future works are a POLC extension to insert a high abstract level structure of language and a hardware implementation of ObAM using commercial of-the-shelf components.

6 Acknowledgements

The author wishes to thank the Computing Science and Correctness of Real Time Systems Group of CNR IASF-MI for the useful discussions. In particular the author thanks prof. G. Sechi for enlightening suggestions and discussions, and prof. P. Guazzoni for the continuous assistance.

7 References

(1) R.Brun, F.Rademakers, Nucl. Instrum. Methods A, **389**, pp. 81-86, (1997).

(2) B.Franek, C.Gaspar, IEEE Trans. Nucl. Sci., **45**, pp. 1946-1950, (1998).

(3) G.De Nardo et al., Nucl. Instrum. Methods A, **471**, pp. 389-402, (2001).

(4) I.Sommerville, "*Software Engineering, Sixth Edition*", Addison – Wesley, (2001).

(5) M. Choban, D. Carvell, "*NASA Guidebook for safety Critical Software*", (1996).

(6) ESA – ESTEC "*TOS-EM, Mathematics and Software Division*", (2001).

(7) Gruppo di Lavoro ASI per le Tecnologie dell'Informazione, "*Ricerca Scientifica e Tecnologica nel settore delle Tecnologie dell'Informazione di interesse spaziale*", (Roma 2001).

(8) M. Abadi, L. Cardelli, "*A theory of object*", Springer, (1996).

(9) B. Liskov, J. Guttag, *"Abstraction and Specification in Program Development*", The MIT Press, (1986).

(10) B.Nordström et al., "*Programming in Martin-Löf's Type Theory – an introduction*", Claredon Press Oxford, (1990).

(11) B. Liskov, S. Zilles, "*Programming with abstract data types*", in: Proc. of the ACM SIGPLAN symposium on Very high level languages (ed. ACM Press), pp. 50-59, (Santa Monica, California, United States, 1974).

(12) B.Shriver, P.Wegner, "*Research Directions in Object-Oriented Programming*", The MIT Press, (1987).

(13) D. Ungar, R.B.Smith, *"Self: The power of simplicity"*, in: Proc. of the OOPSLA' (ed. ACM Press), pp. 227-242, (Orlando, Florida, United States, 1987).

(14) C. Chambers, Cecile Group, *"The Cecil Language, Specification and Rationale"*, Department of Computer Science and Engineering, University of Washington, Seattle, (1998).

(15) M. Kolling, J. Rosenberg, *"Blue Language Specification"*, Monash University, Department of Computer Science and Software, 1997.

(16) B. Ekel, "*Thinking in C++, 2nd Edition*", Pretince Hall, (2001).

(17) M.Sassi, Tesi: "Definizione di una macchina astratta ad oggetti conforme alle linee guida adottate dalle agenzie spaziali e relative al progetto di architetture di calcolo", Dipartimento di Fisica – Università degli Studi di Milano, (2003).

(18) V.T. Jordanov, G.F. Knoll, Nucl. Instrum. Methods A, **345**, pp. 337-345, (1994)

(19) M.Sassi, G.Sechi, *private communication and to be published.*