Journal of Software Engineering and Applications, **2012, 5, 695-710**
http://dx.doi.org/10.4236/jsea.2012.59083 Published Online September 2012 (http://www.SciRP.org/journal/jsea)

Notification Oriented and Object Oriented Paradigms Comparison via Sale System

Jean M. Simão[1,2], Danillo L. Belmonte[1], Adriano F. Ronszcka[2], Robson R. Linhares[1,2], Glauber Z. Valença[2], Roni F. Banaszewski[1], João A. Fabro[2], Cesar A. Tacla[1,2], Paulo C. Stadzisz[1,2], Márcio V. Batista[2]

[1]Graduate School in Electrical Engineering & Industrial Computer Science (CPGEI), Federal University of Technology-Paraná, Curitiba, Brazil; [2]Graduate School of Applied Computing (PPGCA), Federal University of Technology-Paraná, Curitiba, Brazil.
Email: jeansimao@utfpr.edu.br, linhares@utfpr.edu.br, fabro@utfpr.edu.br, tacla@utfpr.edu.br, stadzisz@utfpr.edu.br

Received July 8th, 2012; revised August 11th, 2012; accepted August 23rd, 2012

ABSTRACT

This paper presents a new programming paradigm named Notification-Oriented Paradigm (NOP) and analyses the performance aspects of NOP programs by means of an experiment. NOP provides a new manner to conceive, structure, and execute software, which would allow causal-knowledge organization and decoupling better than standard solutions based upon current paradigms. These paradigms are essentially Imperative Paradigm (IP) and Declarative Paradigm (DP). In short, DP solutions are considered easier to use than IP solutions due to the concept of high-level programming. However, they are considered slower in execution and lesser flexible in development. Anyway, both paradigms present similar drawbacks such as redundant causal-evaluation and strongly coupled entities, which decrease software performance and processing distribution feasibility. These problems exist due to an orientation to a monolithic inference mechanism based on sequential evaluation searching on passive computational entities. NOP proposes another way to structure software and make its inferences, which is based on small, collaborative, and decoupled computational entities whose interaction happens through precise notifications. This paper presents a quantitative comparison between two equivalent implementations of a sale system, one developed according to the principles of Object-Oriented Paradigm (OOP/IP) in C++ and other developed according to the principles of NOP based on a NOP framework in C++. The results showed that NOP implementation obtained quite equivalent results with respect to OOP implementation. This happened because the NOP framework uses considerable expensive data-structures over C++. Thus, it is necessary a new compiler to NOP in order to actually use its potentiality.

Keywords: Notification Oriented Paradigm; Notification Oriented Inference; NOP and OOP Comparison

1. Introduction

This section mentions the drawbacks of the main programming paradigms, introduces a new paradigm, and presents the paper objectives.

1.1. Review Stage

The computational processing power has grown each year and the tendency is that technology evolution contributes to the creation of still faster processing technologies [1,2]. Even if this scenario is positive in terms of pure technology evolution, in general it does not motivate information-technology professionals to optimize the use of processing resources when they develop software [1,3].

This behavior has been tolerated in standard software development where there is no need of intensive processing or processing constraints. However, it is not acceptable to certain software classes, such as software for embedded systems. Such systems normally employ less-powerful processors due to factors such as constraints on power consumption and system price to a given market [1,4,5].

Besides, misuse of computational power in software can also cause overuse of a given standard processor, implying in execution delays [1,4,6]. Still, in complex software, this can even exhaust a processor capacity, demanding faster processor or even some sort of distributions (e.g. dual-core) [1,4,7]. Indeed, an optimization-oriented programming could avoid such drawbacks and related costs [1,4,8].

Thus, suitable engineering tools for software development, namely programming languages and their environments, should facilitate the development of optimized

and correct code [1,9-12]. Otherwise, the engineering costs to produce optimized-code could exceed those of upgrading the processing capacity [1,4,9-11].

Still, suitable tools should also make the development of distributable code easy once, even with optimized code, distribution may be actually demanded in some cases [1,13-16]. However, distribution is itself a problem once, under different conditions, it could entail a set of (related) problems, such as complex load balancing, communication excess, and hard fine-grained distribution [1,4,13,14,17].

In this context, a problem arises from the fact usual programming languages (e.g. Pascal, C/C++, and Java) present no real facilities to develop optimized and really distributable code, particularly in terms of fine-grained decoupling of code [1,3,4,17,18]. This happens due to the structure and execution nature imposed by their paradigms [1,7,9].

1.2. Imperative and Declarative Programming

Usual programming languages are based on the Imperative Paradigm, which cover sub-paradigms such as Procedural and Object Oriented ones [1,10,19,20]. Besides, the latter is normally considered better than the former due to its richer abstraction mechanism. Anyway, both present drawbacks due to their imperative nature [1,10, 19,21].

Essentially, Imperative Paradigm imposes loop-oriented searches over passive elements related to data (e.g. variables, vectors, and trees) and causal expressions (*i.e.* if-then statements or similar) that cause execution redundancies. This leads to the creation of programs as monolithic entities comprising prolix and coupled code, generating non-optimized and interdependent code execution [1,9,21,22].

The Declarative Paradigm is the alternative to the Imperative Paradigm. Essentially, it enables a higher level of abstraction and easier programming [1,20,21]. Also, some declarative solutions avoid many execution redundancies in order to optimize execution, such as Rule Based System (RBS) based on Rete or Hal algorithms [1,23-26]. However, programs constructed using usual languages from Declarative Paradigm (e.g. LISP, PROLOG, and RBS in general) or even using optimized solution (e.g. Rete-driven RBS) also present drawbacks [1, 8,9].

Declarative Paradigm solutions use computationally expensive high-level data structures causing considerable processing overheads. Thus, even with redundant code, Imperative Paradigm solutions are normally better in performance than Declarative solutions [1,10,27]. Furthermore, similarly to the Imperative Paradigm programming, the Declarative one also generates code coupling due to similar search-based inference process [1,4, 21]. Still, other approaches, such as event-driven and functional programming, do not solve these problems even if they may reduce some problems, like redundancies [1,22,27].

1.3. Development Issues & Solution Perspective

As a matter of fact, there are software development issues in terms of ease composition of optimized and distributable code. Therefore, this prompts for new solutions to make simpler the task of building better software. In this context, a new programming paradigm, called Notification Oriented Paradigm (NOP), was proposed regarding some of the highlighted problems [1,4,8,9].

The NOP basis was initially proposed by J. M. Simão as a manufacturing discrete-control solution [28,29]. This solution was evolved as general discrete-control solution and then as a new inference-engine solution [4], attaining finally the form of a new programming paradigm [8-10]. Since then, efforts have been produced to contribute to the establishment of this paradigm [1,30-35].

The essence of NOP is its inference process based on small, smart, and decoupled collaborative entities that interact by means of precise notifications [1,4]. This solves redundancies and centralization problems of the current causal-logical processing, thereby solving processing misuse and coupling issues of current paradigms [1,4,8,9].

1.4. Paper Context and Objective

This paper discusses NOP as a solution to certain current paradigm deficiencies. Particularly, the paper presents a performance study, in a mono-processed case, related to a program based on NOP compared against an equivalent program based on Imperative/Object-Oriented Paradigm.

The NOP program is elaborated in the current NOP framework over C++, whereas the OOP program is elaborated in C++. Thus, an objective of the paper is evaluated the current NOP materialization in terms of performance.

Furthermore, this paper presents the results from two implementations of the NOP Framework. The first is the original one, designed and presented in [10]. The second is an optimized version presented in [34].

Thus, another objective is demonstrate how relevant and appropriated are some refactoring and optimizations in the NOP framework, thereby realizing if is or not necessary to built a NOP compiler.

2. Background

This section explores programming paradigm drawbacks.

2.1. Imperative Programming Issues

The main (related) drawbacks of Imperative Programming are concerned to redundancies and code coupling. The 1st mainly affects the processing time and the 2nd the processing distribution, as detailed in the next subsections [1,4].

2.1.1. Imperative Programming Redundancy

In Imperative Programming, like procedural or object oriented programming, a number of code redundancies and interdependences comes from the manner the causal expressions are evaluated. This is exemplified in the pseudo-code in **Figure 1** that represents a usual code elaborated without strong technical and intellectual efforts. This means that the pseudo-code was elaborated in a non-complicated manner, as software elaboration should ideally be [1,8,10].

In the example, each causal expression has three logical premises and a loop forces the sequential evaluation of all causal expressions. However, most evaluations are unnecessary because usually just few attributes of objects (*i.e.* variables) have their values changed at each iteration. This type of code causes the problem called, in the computer science, temporal and structural redundancy [1, 4,25].

The temporal redundancy is the repetitive unnecessary evaluation of causal expressions in the presence of element states (e.g. attribute or variable states) already evaluated and unchanged. For instance, this occurs in the considered loop-oriented code example. The structural redundancy, in turn, is the recurrence of a given logical expression evaluation in two or more causal expressions. For instance, the logical expression (*object*_1.*attribute*_1 = 1) is replicated in several causal expressions (*if-then statements*) [1,4,8].

These redundancies can be considered unimportant in this didactic code example, mainly if the number (*n*) of

```
1 ...
2 while (true) do
3   if((object_1.attribute_1 = 1) and
4      (object_2.attribute_1 = 1) and
5      (object_3.attribute_1 = 1))
6   then
7       object_1.method_1();
8       object_2.method_1();
9       object_3.method_1();
10  end_if
11  ...
12  if((object_1.attribute_1 = 1) and
13     (object_2.attribute_n = n) and
14     (object_3.attribute_n = n))
15  then
16      object_1.method_n();
17      object_2.method_n();
18      object_3.method_n();
19  end_if
20 end_while
21 ...
```

Figure 1. Example of imperative code [1].

causal expressions is small. However, if more complex examples were considered, integrating many (remaining) redundancies, there would be a tendency to performance degradation and increasing of development complexity inclusively to avoid that degradation [1,8,10].

The code redundancies may result, for example, in the need of a more powerful processor than it is really required [1,4,7]. Also, they may result in the need for code distribution to processors, thereby implying in other problems such as module splitting and synchronization. These problems, even if solvable, are additional issues in the software development whose complexity increases as much as the fine-grained code distribution is demanded, particularly in terms of logical-causal ("*if-then*") calculation [1,4,7,9].

2.1.2. Imperative Programming Coupling

Besides the usual repetitive and unnecessary evaluations in the imperative code, the evaluated elements and causal expressions are passive in the program decisional execution, although they are essential elements in this process. For instance, a given if-then statement (*i.e.* a causal expression) and concerned variables (*i.e.* evaluated elements) do not take part in the decision with respect to the moment in time they must be evaluated [1,4].

The passivity of causal expressions and concerned elements is due to the way they are evaluated in the time. An execution line in each program (or at least in each program thread) carries out this evaluation, usually guided by means of a set of loops. As these causal expressions and concerned elements do not actively conduct their own execution (*i.e.* they are passive), their interdependency is not explicit in each program execution [1,4].

Thus, at first, causal expressions or evaluated elements depend on results or states of others. This means that they are somehow coupled and should be placed together, at least in the context of each module. This coupling increases code complexity, which complicates, for instance, an eventual distribution of each single code part in fine-grained way. This makes each program module, or even the whole program, a monolithic computational unit [1, 4].

2.1.3. Imperative Programming Distribution Hardness

When distribution is intended (process, processor, and cluster distribution), a code analysis could identify less dependent code sets to facilitate splitting. However, this is normally a complex activity due to the code coupling and complexity caused by imperative programming [1,18, 36].

In this sense, well-designed software composed of modules as decoupled as possible, using advanced and

quite complicated software engineering concepts like *aspects* [13] and *axiomatic design* [37], can help distribution. Still, middleware such as CORBA and RMI would be helpful in terms of infrastructure to some types of module distribution, if there is enough module decoupling [1,13,38,39]. In spite of those advances, distribution of single code elements or even code modules is still a complex activity demanding research efforts [1, 13,14,17,36,40]. It would be necessary additional efforts to achieve easiness in distribution (e.g. automatic, fast, and real-time distribution), as well as correctness in distribution (e.g. balanced and minimal inter-dependent distribution) [1,4].

Indeed, distribution hardness is an issue once there are contexts where distribution is actually necessary [1,7,15, 16]. For instance, a given optimized program exceeding the capacity of a given processor would demand processing splitting [1,6]. Other instances are programs that must guaranty error isolation or even robustness by distributed module redundancy [1,28]. These features can be found in application such as nuclear-plant control [41], intelligent manufacturing [29,42,43], and cooperative controls [44].

Besides, there are other applications that are inherently distributed and need flexible distribution, such as those of ubiquitous computing. More precise examples are sensor networks and some intelligent manufacturing control [1,43,45]. Moreover, the easy and correct distribution is an expectation due to the reduction of processor prices and the communication networks advances as well [1, 10,46].

2.1.4. Imperative Programming Development Hardness

In short, as explained in terms of Imperative and Declarative Paradigms, current paradigms do not make easy to achieve the following qualities together [1]:

- Effective (causal) code optimization to be sure about the eventual need of a faster processor or multiprocessing.
- Easy way to compose correct code (*i.e.* without errors).
- Easy code splitting and distribution to processing nodes.

This is a problem mainly when the increasing market demand by software is considered, where development easiness, code optimization, and processing distribution are current requirements [1,47-49]. This software development "crisis" impels new researches and solutions to make simpler the task of building better software [1].

In this context, a new programming paradigm called Notification Oriented Paradigm (NOP) was proposed to solve some of the highlighted problems. NOP keeps the main advantages of Declarative Programming/Rule Based Systems (e.g. higher causal abstraction and organization by means of fact base and causal base) and Imperative/ Object Oriented Programming (e.g. reusability, flexibility, and suitable structural abstraction via classes and objects). Moreover, NOP would evolve some of their concepts and solve some of their deficiencies [1,4,8,10].

2.2. Declarative Programming Issues

A well-known example of Declarative Programming and its nature is Rule Based System (RBS) [1,4,50]. A RBS provides a high-level language in the form of causal-rules, which prevents the developers from algorithm particularities [1,50]. RBS is composed of three general modular entities (Fact Base, Rule Base, and Inference Engine) with well-distinguished responsibilities, as usual in declarative language (e.g. LISP, PROLOG, and CLIPS) [1,51].

In Declarative Programming, the variable states are dealt in a Fact Base and the causal knowledge in a Causal Base (Rule Base in RBS), which are automatically matched by means of an Inference Engine (IE) [1,24,50]. Moreover, some IE algorithms (e.g. RETE [23-25], TREAT [52,53], LEAPS [54], and HAL [26] algorithms) avoid most of temporal and structural redundancies [1,10]. However, the data structures used to solve redundancies in those IEs implies in too much consuming of processing capacity [1,25].

Actually, the use of Declarative Programming only compensates when the software under development presents many redundancies and few data variation. Also, in general, an IE related to a given declarative language limits the inventiveness, makes difficult some algorithm optimizations, and obscures hardware access, which can be inappropriate in certain contexts [1,10,22,27,55].

A solution to these problems can be the symbiotic use of Declarative and Imperative Programming [1,19,55]. Such approach has been presented, like CLIPS++, ILOG Rules, and R++. However, they would not be popular due to factors like syntax and paradigms mixing or technical cultural reasons [1,10]. Anyway, even Declarative Programming being a relevant solution, it does not solve certain issues [1].

Indeed, beyond processing-overhead, Declarative Programming also presents code coupling. Each declarative program has also an execution or inference policy whose essence is a monolithic entity (e.g. Inference Engine) responsible for analyzing every passive data-entity (Fact-Base) and causal expression (Causal-Base). Thus, the inference, based upon a search technique (*i.e.* matching), implies a strong dependency between facts and rules because they together constitute the search space [1,4].

2.3. Other Programming Approach Drawbacks

Enhancements in the context of Imperative and Declara-

tive Paradigm have been provided to reduce the effects of recurrent loops or searches, such as event-driven programming and functional programming [1,10,51,56]. Event programming and functional programming have been used to different software such as discrete control, graphical interfaces, and multi-agent systems [1,10,51, 56].

Essentially, each event (a button pressing, a hardware interruption or a received message) triggers a given execution (process, procedure or method execution), usually in a given sort of module (block, object or even agent), instead of repeated analysis of the conditions for its execution.

The same principle applies to the called functional programming whose difference would be function calling via other function in place of events. Still, function means procedure, method or similar unity. Besides, functional and event programming used together would be usual [1].

However, the algorithm in each module process or procedure is built using Declarative or Imperative programming. This implies in the highlighted deficiencies, namely code redundancy and coupling, even if they are diminished by events or function calls. Indeed, if each module has extensible or even considerable causal-logical calculation, they can be a problem together in terms of processing misuse and distribution. This may demand special design effort to achieve optimization and module decoupling [1].

An alternative programming approach is the Data Flow Programming that supposedly should allow the program execution oriented by data instead of an execution line based on search over data. Therefore, this would allow decoupling and distribution [1,14]. The distribution in Data Flow Programming is achieved in arithmetical processing, however it is not really achieved in logical-causal calculation [1,14,17]. This calculation happens by current advanced inference engines, namely Rete [1,17,57].

The fact is current inference engines attempt to achieve a data-driven approach. However, the inference process is still based on searches even if they use data from (some sort of) object-oriented tree to speed up the inference cycle or searches. Thus, the highlighted problems remain [1].

2.4. Enhancement in Programming

In short, as explained in terms of Imperative and Declarative Paradigms, current paradigms do not make easy to achieve the following qualities together [1]:

- Effective (causal) code optimization to be sure about the eventual need of a faster processor or multiprocessing.
- Easy way to compose correct code (*i.e.* without errors).
- Easy code splitting and distribution to processing nodes.

This is a problem mainly when the increasing market demand by software is considered, where development easiness, code optimization, and processing distribution are current requirements [1,47-49]. This software development "crisis" impels new researches and solutions to make simpler the task of building better software [1].

In this context, a new programming paradigm called Notification Oriented Paradigm (NOP) was proposed to solve some of the highlighted problems. NOP keeps the main advantages of Declarative Programming/Rule Based Systems (e.g. higher causal abstraction and organization by means of fact base and causal base) and Imperative/Object Oriented Programming (e.g. reusability, flexibility, and suitable structural abstraction via classes and objects). Moreover, NOP would evolve some of their concepts and solve some of their deficiencies [1,4,8,10].

3. Notification Oriented Paradigm (NOP)

The Notification Oriented Paradigm (NOP) introduces a new concept to conceive, construct, and execute software. NOP is based upon the concept of small, smart, and decoupled entities that collaborate by means of precise notifications to carry out the software inference [1,4,8]. This would allow enhancing software applications performance and potentially makes easier to compose software, both non-distributed and distributed ones [1,10].

3.1. NOP Structural View

NOP causal expressions are represented by common rules, which are naturally understood by programmers of current paradigms. However, each rule is technically enclosed in a computational-entity called "Rule" [1,8]. In **Figure 2**, there is a Rule content example, which would be related Sale System.

Structurally, a Rule has two parts, namely a "Condition" and an "Action", as shown by means of the UML class diagram in **Figure 3**. Both are entities that work together to handle the causal knowledge of the Rule computational-entity. The Condition is the decisional part, whereas the Action is the execution part of the Rule. Both make reference to factual elements of the system [1,8,10].

NOP factual elements are represented by means of a special type of entity called "Fact_Base_Element" (FBE). A FBE includes a set of attributes. Each attribute is represented by another special type of entity called "Attribute" [1,8,10].

Attributes states are evaluated in the Conditions of

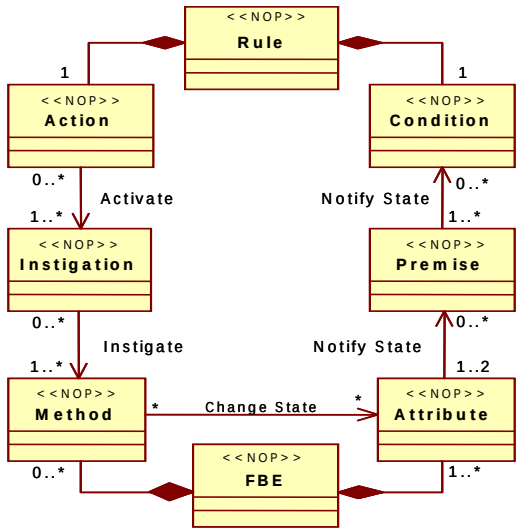

Figure 2. Rule and Fact_Base_Element (FBE) class diagram [1,8,10].

Rules by associated entities called "Premises". In the example, the Condition of the Rule is associated to three Premises, which verify the state of FBE Attributes as follow: 1) Is the product branch perishable? 2) Is the product valid? 3) Is the product perishable date? [1,8,10].

When each Premise of a Rule Condition is true, which is concluded via a given inference process, the Rule becomes true and can activate its Action composed of special-entities called "Instigations". In the considered Rule, the Action "has" only one Instigation that makes the System shows a message that the product is perishable [1,8,10].

Instigations are linked to and instigate the execution of "Methods", which are another special-entity of FBE. Each Method allows executing services of its FBE. Generally, the call of FBE Method changes one or more FBE Attribute states, thereby feeding the inference process [1,8,10].

3.2. NOP Inference Process

The NOP inference process is innovative once Rules have their inference carried out by active collaboration of its notifier entities [4]. In short, the collaboration happens as follow: for each change in an Attribute state of a FBE, the state evaluation occurs only in the related Premises and then only in related and pertinent Conditions of Rules via punctual notifications between the collaborators [1,4,8].

In order to detail this Notification Oriented Inference, it is firstly necessary to explain the Premise composition. Each Premise represents a Boolean value about one or even two Attribute state, which justify its composition: 1) a reference to a discrete value of an Attribute, called *Reference*, which is received by notification; 2) a logical operator, called *Operator*, useful to make comparisons; and 3) another value called *Value* that can be a constant or even a discrete value of other referenced Attribute [1,4,8,10].

A Premise makes a logical calculation when it receives notification of one or even two Attributes (*Reference* and even *Value*). This calculation is carried out by comparing the *Reference* with the *Value*, using the *Operator*. In a similar way, a Premise collaborates with the causal evaluation of a Condition. If the Boolean value of a notified Premise is changed, then it notifies the related Condition set [1,4,8].

Thus, each notified Condition calculates their Boolean value by the conjunction of Premises values. When all Premises of a Condition are satisfied, a Condition is also satisfied and notifies the respective Rule to execute [1,4]. The collaboration between NOP entities via notifications can be observed at the schema illustrated in **Figure 4**. In this schema, the flow of notifications is represented by arrows linked to rectangles that symbolize NOP entities [1,4,8].

An important point to clarify about NOP collaborative entities is that each entity (e.g. Attributes) registers the entities interested in its state (e.g. Premises) in the moment that they are created. For example, when a Premise is created and makes reference to an Attribute, the latter automatically includes the former in its internal set of entities to be notified when its state change [1,4,8].

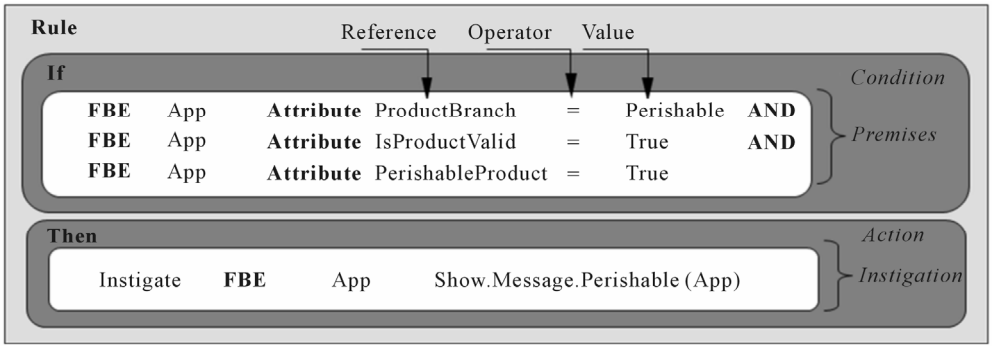

Figure 3. The representation of a rule.

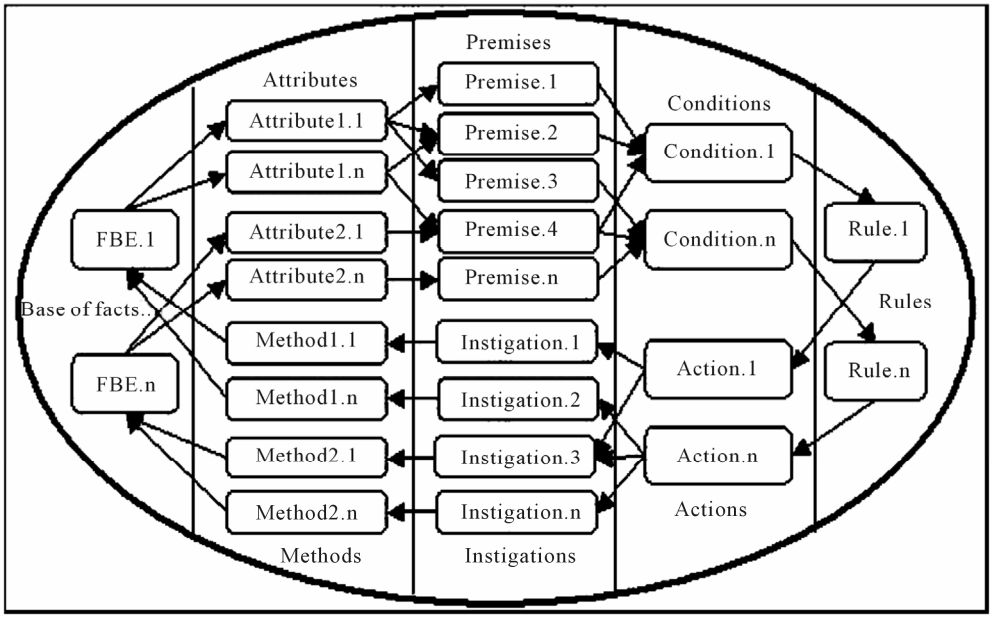

Figure 4. Rule notification chain and their collaborators [1,4,8,10].

3.3. NOP-Redundancy Avoidance—Performance

In NOP, an Attribute state is evaluated by means of a set of logical expression (Premise) and causal expression (Condition) in the changing of its state. Thanks to the cooperation by means of precise notifications, NOP avoids the two types of aforementioned redundancies [1,4,8,10].

The temporal redundancy is solved in NOP by eliminating searches over passive elements, once some data-entities (e.g. Attributes) are reactive in relation to their state updating and can punctually notify only the parts of a causal expression that are interested in the updated state (e.g. Premises), avoiding that other parts and even other causal expressions be unnecessarily (re-)evaluated [1,4, 8,10].

Indeed, each Attribute notifies just the strictly concerned Premise due to state change and each Premise notifies just the strictly concerned Condition due to state change, therefore implicitly avoiding temporal redundancy. Besides, the structural redundancy is also solved in NOP when Premise collaboration is shared with two or more causal expressions (*i.e.* Conditions). Thus, the Premise carries out logic calculation only once and shares the logic result with the related Conditions, thereby avoiding re-evaluations [1,4,8,10].

3.4. NOP—Decoupling and Distribution

Actually, besides solving redundancy and then performance problems, NOP also is potentially applicable to develop parallel/distributed applications because of the "decoupling" (or minimal coupling to be precise) of entities. In inference terms, there is no great difference if an entity is notified in the same memory region, in the same computer memory or in the same sub-network [1,4,8,10].

For instance, a notifier entity (e.g. an Attribute) can execute in one machine or processor whereas a "client" entity (e.g. a Premise) can execute in another. For the notifier, it is only necessary to know the address of the client entity. However, these issues also should be considered in more technical and experimental details in future publications once there are current works in this context [1,4,8,10].

3.5. NOP Originality

At first, NOP entities (Rules and FBEs) may be confused as just an advance of Rule Based, Object Oriented, and Event-Driven Systems, including then Data-Flow-like Programming and Inference Engines. However, NOP is far than a simple evolution of them. It is a new approach that proposes Rule and FBE smart-entities composed of other collaborative punctual-notifier smart-entities, which provide new type of logical-causal calculation or inference process [1,4,8,10].

This inference solution, in turn, is not just an application of known software notifier patterns, useful to Event-Driven Systems, such as the *observer-pattern.* It is the extrapolation of that once the execution of the NOP logical-causal calculation via punctual notifications has not been conceived before. At least, this is the honest authors' perception after more than one decade of literature reviewing [1,4,8,10].

Indeed, this inference innovation changes all the software essence with respect to logical-causal reasoning (*i.e.*

one of its essential parts) and then makes the solution a new programming paradigm. Moreover, as NOP changes the form in which software is structured and executed, it also determines a change in the form that software is conceived [1,4,8,10].

3.6. NOP Implementation

In order to provide the use of NOP, its entities were materialized in C++ language in the form of a framework [10]. Indeed, it is usual to emergent paradigms be materialized by means of programming languages of current paradigms, already changing them somehow, before the conception of a particular language and compiler [10].

Anyway, developed NOP applications have been made just by instantiating the framework [10,31-33,35]. Moreover, to make easier this process, a prototypal wizard tool has been proposed to automate this process. It is a tool that generates NOP smart-entities from rules elaborated in a graphical interface. In this case, developers "only" need to implement FBEs with Attributes and Methods, once other NOP special-entities will be composed and linked by the tool. This allows using the time to the construction of the causal base (*i.e.* composition of NOP rules) without concerns about instantiations of the NOP entities.

4. The Sale Order System

In order to make some comparisons between NOP and Object-Oriented Paradigm (OOP), this paper presents a Sale Order System as case of study. This application was firstly described (in Portuguese and in shorter way) in [32] and was there first used to some other previous experiments. This system was proposed due to the relevance of its nature.

Indeed, there are many systems of the same domain built in OOP. Thus, the creation of a NOP version is interesting. Among all the applications designed in NOP so far, there is none with the same commercial focus and from the same domain.

4.1. Sale Order System: Requirements

This relevant proposed software system, thought to compare NOP and OOP implementations, presents the requirements considered in **Tables 1-3**.

The software that has been used in all comparisons in this paper implements the functional requirement number RF1 from **Table 1**, the sub-requirements from **Table 2** (related to RF1), and all non-functional requirements from **Table 3**.

Table 1. Functional requirements.

RF1	To provide selling of products.

Table 2. Sub-requirements related to functional requirements.

SR1.1	To not allow selling products with stock equal zero.
SR1.2	To debit the stock of the sold quantity of product.
SR1.3	To allow selling more than one product for each sale.
SR1.4	To persist the data of the sale.
SR1.5	To not allow selling perishable product expired.

Table 3. Non-functional requirements.

RNF1	To be implemented using C++ language.
RNF2	To be implemented on both paradigms, OOP in C++ and NOP Framework over C++.
RNF3	To run under console (ms-dos) environment.
RNF4	To persist data in text files.

4.2. Sale Order System: Structure

The requirement definition allows modeling the system by means of a class diagram shown in **Figure 5**, which is useful to both OOP and NOP implementations.

The SalesOrder class is the most important, having association with Customer, PaymentForm, and SalesOrderItensList. Moreover, it is possible to verify that the Product class is specialized in three different types of products.

4.3. Sale Order System: Dynamics

The **Figure 6** shows an Activity Diagram of the entire selling process without focusing on any implementation.

4.4. Sale Order System: Execution

The sale starts with customer code, which passes by a verification to check if it is valid. After that, it is necessary to inform the branch of the product, as well as check its veracity. The sale process continues requesting products which will compose the sale order.

The system provides validations for product and customer existence. Moreover, the system checks if the product is available in the stock. If the chosen product belongs to the perishable branch, the system evaluates its expiration date. If the product life is expired, it cannot be sold.

After the whole cycle of product insertion into the sale order, the sale can be completed filling out the payment form. There are only two payment forms available, which are cash and installment payment. If the payment chosen is installment, the system checks if the customer can complete the purchase, comparing his credit limit

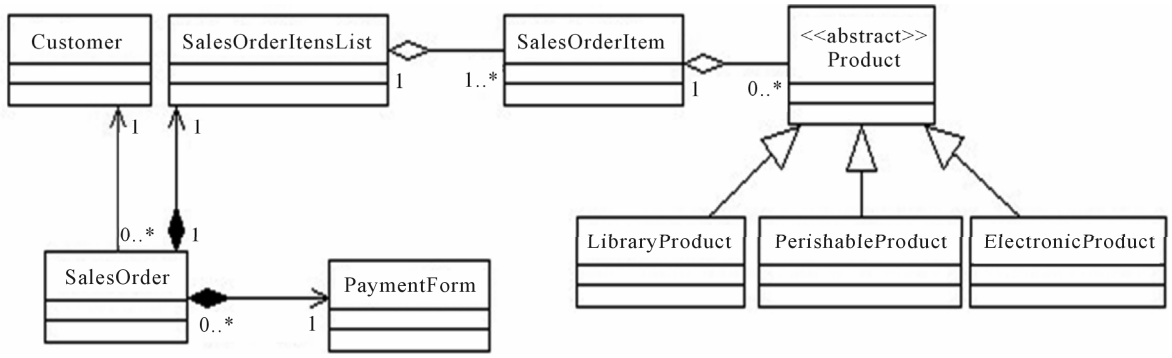

Figure 5. Class diagram of sale order system.

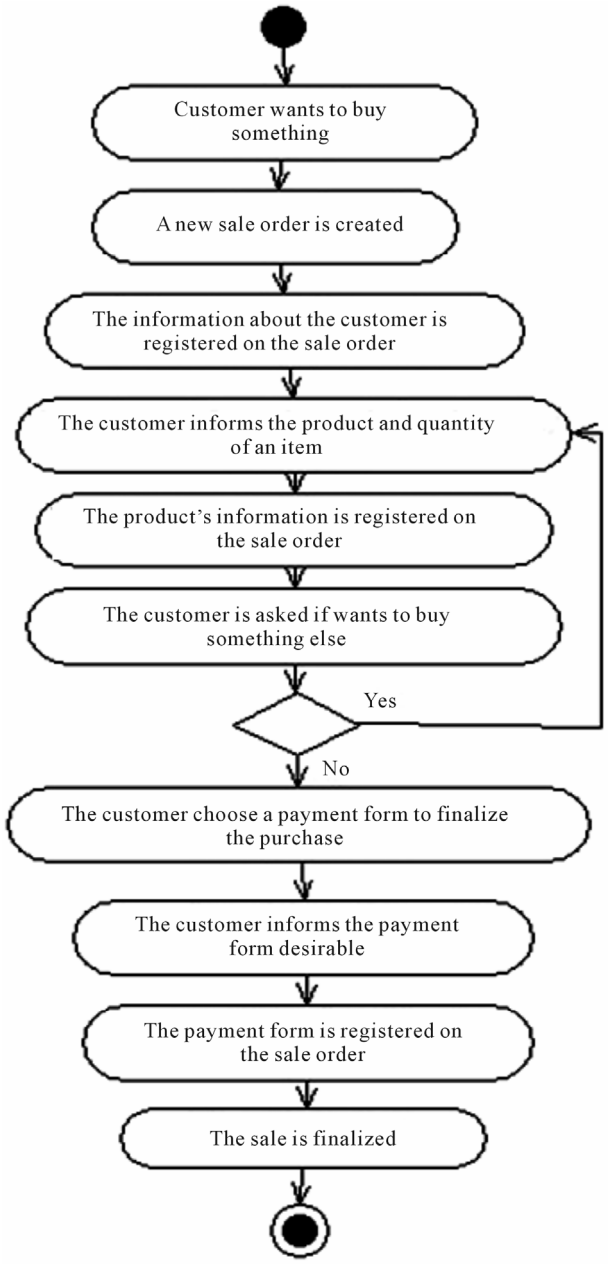

Figure 6. Activity diagram of the sale.

with the total value of the sale. Actually, the system has information about the credit limit of the customer.

Still, on the customer's profile there is a parameter that provides a classification type. This classification type is used to provide a special discount during the sale completion. There is a range of 20 different customer's classification types, which can provide discounts from 5% to 95%.

5. A Performance Study

This section shows the implementation of OOP (C++) and in an equivalent manner in NOP (NOP framework in C++) of the Sale System. Still, the section presents comparative tests performed between these two implementations.

5.1. Implementation Details

Both OOP and NOP implementations use same classes, methods and procedures. The difference between them is the fact that the principal method of the whole system (*i.e.* Sale method) has been rewritten for each version.

In NOP, the sale method has its code written under the principles of Rules and other collaborator smart-entities. The **Figure 7** shows an example of a code written in OOP (in C++) and NOP (in NOP Framework over C++).

There are no more *if-then* causal tests and nested code in NOP version. The entire sale flow is governed by Rules and other collaborator smart-entities. For each NOP Attribute that has its state changed, it will start the notification process.

All causal tests performed in the OOP version are handled quite differently in the NOP version. The **Figure 7** somehow shows the differences between them. In turn, the **Figure 8** shows the pseudo-code of the process to create a sale order based on the OOP paradigm.

5.2. First Experiments and Results

The experiments were performed using the original version of the NOP Framework, *i.e.* 1st stable version designed


```
if (productBranch == "1")     { objProduct = new EletronicProduct(); }
else if(productBranch == "2"){ objProduct = new LibraryProduct();    }
else if(productBranch == "3"){ objProduct = new PerishableProdcut();}
```

```
RuleObject* ruleElectProd = new RuleObject("Rule5", scheduler, Condition::CONJUNCTION);
ruleElectProd->addPremise(new Premise(app->attProductBranch,1,Premise::EQUAL));
ruleElectProd->addInstigation(new Instigation(app->mtInstantiateElectronicProduct));
ruleElectProd->end();

//Create an library product
RuleObject* ruleLibProd = new RuleObject("Rule6", scheduler, Condition::CONJUNCTION);
ruleLibProd->addPremise(new Premise(app->attProductBranch,2,Premise::EQUAL));
ruleLibProd->addInstigation(new Instigation(app->mtInstantiateLibraryProduct));
ruleLibProd->end();

//Create an perishable product
RuleObject* rulePerishProd= new RuleObject("Rule7", scheduler, Condition::CONJUNCTION);
rulePerishProd->addPremise(new Premise(app->attProductBranch,3,Premise::EQUAL));
rulePerishProd->addInstigation(new Instigation(app->mtInstantiatePerishableProduct));
rulePerishProd->end();
```

Figure 7. Equivalent code in OOP and NOP.

by Banaszewski [9]. Tests performed with optimized version of the Framework have also been added in order to verify the performance difference between Frameworks.

The optimized version was developed by NOP research group [9]. The same set of parameters was used in all experiments. This set of parameters was implemented for the version of the system developed purely on the principles of OOP and also for the version designed in NOP.

It has been created 100, 1000, 10,000, and 100,000 sales for the test cases. The experiment was performed three times to ensure the accuracy of the data. All the data from the Sale Orders which were used in the following experiments were inserted directly on the code. This kind of implementation is well known as hard-coding. The **Tables 4-6** summarize the data which each sale order used during each experiment cycle.

The experiments were performed on a PC with Core 2 Duo processor with 3 Gigabytes of RAM. Tests were performed on Windows and Linux operating systems. Both operating systems are preemptive multitasking, although Linux had better time response of the processor. For the Windows experiment the Visual Studio 2008 was used with its C++ native compiler to design the system. In Linux experiment the GCC compiler was used.

The results of the first test experiments are shown in **Tables 7** and **8**. The first column in **Table 7** represents

Table 4. First test sale order content.

First Experiment Order
Customer
• Customer #1
Products
• Product #1—Electronic type
Payment Form
• Cash

Table 5. Second test sale order content.

Second Experiment Order
Customer
• Customer #20—Type number 20
Products
• Product #1—Electronic type
• Product #2—Electronic type
• Product #3—Electronic type
• Product #4—Electronic type
• Product #5—Electronic type
Payment Form
• Cash
Discount
• 95% discount for customers of type 20.

Table 6. Third test sale order content.

Third Experiment Order
Customer
• Customer #20—Type number 20
Products
• Product #6—Perishable type (Shelf date expired)
• Product #7—Perishable type
• Product #8—Perishable type
• Product #9—Perishable type
• Product #10—Perishable type
• Product #11—Perishable type
Payment Form
• Installment (Customer does not have sufficient credit to use this payment form)
• Cash
Discount
• 95% discount for customers of type 20.


```
begin
    create an object sale order
    while (customer is empty or invalid)
    begin
        the application requests a customer
        if (the customer chosen does not exists)
        begin
            return invalid customer message
        end
    end

    add into the sale order a reference to the costumer
    while (the product is empty or invalid)
    begin
        while (branch of product is empty or invalid)
        begin
            the application requests a product branch
            if (product branch is different of 1,2 or 3)
            begin
                return invalid product branch message
            end
        end

        if (product branch = 1)
        begin
            create an electronic product
        end

        else if(product branch = 2)
        begin
            create a library product
        end

        else if (product branch = 3)
        begin
            create a perishable product
        end

        the application requests a product code
        if (the product chosen does not exist)
        begin
            return invalid product message
        end

        if (product branch = 3)
        begin
            if (product validity date is minor than actual date)
            begin
                return invalid and expired product
            end
        end

        the application requests the quantity of product that will be sold
        if (sold quantity is minor than the product stock)
        begin
            return invalid product and product without enough stock to be sold
        end

        create a new object item for the Sale Order which has reference for the sold product
        add the reference of the sold item into the sale order itens list
        if (the costumer wants to inform more products)
        begin
            return invalid product message. This will fire a new request of products
        end
    end

    while (payment form is empty or invalid)
    begin
        the application requests a payment form
        if (payment form chosen is different of cash or installment payment)
        begin
            return invalid payment form message
        end

        if (payment form is installment payment)
        begin
            if(customer credit is minor than the total of sale)
            begin
                return invalid payment form message and insufficient credit.
            end
        end
    end

    add into the sale order a reference for the payment form
    if (customer type = 0)
    begin
        the sale order will not have any discount
    end
    else if (customer type = 1)
    begin
        the sale order will have 5% discount
    end
    ...
    else if (customer type = 20)
    begin
        the sale order will have 95% discount
    end

    save the sale order into the database file
end
```

Figure 8. OOP pseudo code of the selling function.

Table 7. First test experiment on Windows environment.

Total of sale orders	OOP	NOP (Original)	NOP (Optimized)	Performance NOP over OOP
100	208	280	231	−11.10%
1000	2028	2974	2302	−13.51%
10,000	22,734	30,019	24,025	−5.67%
100,000	234,650	338,390	243,226	−3.65%

Table 8. First test experiment on Linux environment.

Total of sale orders	OOP	NOP (Original)	NOP (Optimized)	Performance NOP over OOP
100	98	106	100	−2.04%
1000	917	1020	965	−5.23%
10,000	8935	9775	9552	−6.90%
100,000	90,491	98,668	94,559	−4.49%

the numbers of sale orders that were created. The column OOP shows the results of the elapsed time in milliseconds during the tests, as well as the column NOP (original) and NOP (optimized). The last column shows the percentage difference between NOP Optimized and OOP version.

Besides, a first version of this experiment concerning **Table 7** was done and presented in [32], where it was considered just OOP and NOP original versions of the Sale System. Still, the case with 100,000 sales orders was not taken into account. The same environment was used and the tests had almost the same results here presented.

The results in **Tables 7** and **8** shows that the system purely written in OOP obtained a little less execution time when compared with the system version written purely in NOP. The scenario created for the first experiment did not allow NOP to explore some OOP problems (i.e. causal and temporal redundancy).

The code organization developed in OOP was constructed in the same way of the pseudo code shown on the **Figure 8**. However, in the considered experiment, the OOP code does not activate the temporal redundancy shown from line 85 to 97. Thus, this code does not perform unnecessary tests and iterations.

Moreover, all the sale orders created are composed of only one product. Many causal evaluations that verify the product's shelf life in a sale order were not fully explored because all the information contained in those sale orders were valid (*i.e.* customer, product, payment form).

5.3. Second Experiments and Results

In the second experiment, there are more evaluations performed. In this experiment, each sale order has a total of five products sold. Also, there is the indicator of customer's classification type with the range of 20 types. This indicator provided a special discount when the sale order was finalized. This verification was implemented in the OOP version in a nested way (*i.e.* If-else-if).

In the **Figure 9**, it is possible to observe the representation of temporal redundancy in OOP and how NOP implements the same code using. With OOP version, it was possible to create a typical temporal redundancy in order to compare its impact when compared to NOP. In **Tables 9** and **10**, it is presented the results of the second stage of the experiments.

Besides, a similar version of this experiment concerning **Table 9** was done and shortly presented in [34], where it was considered just NOP Original and NOP Optimized versions of the Sale System. Still, the case with 100,000 sales orders was not taken into account. Almost the same environment was used and the tests had quite similar results to those here presented.

From the results in **Tables 9** and **10**, it was possible to verify results of a scenario with temporal redundancy in OOP implementation. In this context, for each iteration, it totalizes a total of 19 unnecessary evaluations.

According to the results presented in **Tables 9** and **10**, in general, the system written in OOP obtained an execution time slightly better than NOP version. However, the **Table 9** shows that in the Windows version, NOP obtained from 7 to 10 percent better results when performed 10,000 and 100,000 iterations.

5.4. Third Experiment and Results

The **Tables 11** and **12** express results of third experiment where the scenario was changed. The causal evaluations

Table 9. Second test experiment on Windows environment.

Total of sale order	OOP	NOP (Original)	NOP (Optimized)	Performance NOP over OOP
100	780	936	790	−1.22%
1000	8003	9968	8211	−2.59%
10,000	103,688	122,725	93,943	10.37%
100,000	969,848	1,297,993	900,251	7.73%

Table 10. Second test experiment on Linux environment.

Total of sale order	OOP	NOP (Original)	NOP (Optimized)	Performance NOP over OOP
100	379	405	391	−3.16%
1000	3689	3967	3847	−4.28%
10,000	36,836	39,268	38,401	−4.24%
100,000	367,442	393,057	384,161	−4.55%


```
//Giving Discount based on Customer type
if (objCustomer->GetCustomerType() == 0)
{
    discountPercent = 0.0;
}
else if (objCustomer->GetCustomerType() == 1)
{
    discountPercent = 0.5;
}
...
else if (objCustomer->GetCustomerType() == 20)
{
    discountPercent = 0.95;
}
```

```
RuleObject* ruleDescType0 = new RuleObject("Rule18", scheduler, Condition::DISJUNCTION);
ruleDescontoTipo0->addPremise(new Premise(app->attCustomerType, 0, Premise::EQUAL));
ruleDescontoTipo0->addInstigation(app->attDiscountPercentage, 0.0);
ruleDescontoTipo0->end();

RuleObject* ruleDescType1 = new RuleObject("Rule19", scheduler, Condition::DISJUNCTION);
ruleDescType1->addPremise(new Premise(app->attCustomerType, 1, Premise::EQUAL));
ruleDescType1->addInstigation(app->attDiscountPercentage, 0.5);
ruleDescType1->end();
... //There are one rule for each customer type, from 0 to 20
RuleObject* ruleDescType20 = new RuleObject("Rule38", scheduler, Condition::DISJUNCTION),
ruleDescType20->addPremise(new Premise(app->attCustomerType, 20, Premise::EQUAL));
ruleDescType20->addInstigation(app->attDiscountPercentage, 0.95);
ruleDescType20->end();
```

Figure 9. Temporal redundancy in OOP and Rules PON.

Table 11. Third test experiment on Windows environment.

Total of sale order	OOP	NOP (original)	NOP (optimized)	Performance NOP over OOP
100	894	1121	967	−8.16
1000	9645	12,548	10,587	−9.76
10,000	115,252	152,789	134,420	−16.63
100,000	1166,195	1688,341	1214,634	−4.15

Table 12. Third test experiment on Linux environment.

Total of sale order	OOP	NOP (original)	NOP (optimized)	Performance NOP over OOP
100	390	425	405	−3.84%
1000	3899	4212	4036	−3.51%
10,000	38,652	42,645	40,486	−4.74%
100,000	387,716	430,053	405,855	−4.67%

were increased in order to have more evaluations during the execution of experiments. It was entered invalid information (e.g. customer, product, and payment form). Also, wrong identification codes cause unnecessary evaluations that check the validity of information informed.

In **Tables 11** and **12**, the results show that the system developed entirely in NOP obtained similarly runtime results when compared with the system developed in OOP. Even though it was expected a better runtime performance of the NOP solution when compared to the OOP solution. Nevertheless, it is possible to believe in the NOP approach for other systems with more unnecessary evaluations and redundancies than those of the considered experiment.

Still, something that can be noticed in all the comparisons which have been made between Windows and Linux for the same application is that they presented quite different execution times. This may be due to differences in machine code generated by compilers.

5.5. Considerations

Analyzing the difference between the execution time of the original and optimized version of NOP against the OOP version shown in **Tables 2** and **3**, it is possible to infer that the overhead of the NOP Framework implementation did not allow NOP exploring its capabilities.

The framework's core uses native data structures such as pointers and linked lists, which have the role of storing elements that will be notified when it is necessary. With the increase of system elements, more elements should be notified and this overhead could generate exhaustive use of the processing power that will result in no changes as expected.

All the data from the experiments place, in general, OOP version in a better position than NOP with slightly better results. However, these findings are not sufficient to judge OOP system version as superior when compared to NOP version. This is due particularly to the fact that NOP is embodied in the form of a Framework/wizard built over the C ++ programming language.

The experiments performed can conclude that the development of systems designed in NOP is plausible. Although the framework used for the realization of the paradigm needs some improvements, its current state allows the development of functional systems.

6. Conclusion and Future Works

This section discusses NOP properties and future works.

6.1. Notification Oriented Paradigm Features

NOP would be an instrument to improve applications' performance in terms of causal calculation, especially of complex ones such as those that execute permanently and need excellent resource use and response time. This would be possible thanks to the notification mechanism, which allows an innovative causal-evaluation process with respect to those of current programming paradigms [1,8,9,10,29].

The notification mechanism is composed of entities that collaboratively carry out the inference process by means of notifications, supposedly providing solutions to deficiencies of current paradigms [1]. In this context, this paper addressed the performance subject making some comparisons of NOP and Imperative Programming instances.

6.2. NOP Performance

Previous tests implemented in environments with a considerable number of redundancies show that NOP improves performance compared to other approaches, by means of its innovative notification mechanism [1,10]. This was expected once temporal and structural redundancies are avoided by NOP, thereby guaranteeing suitable performance by definition [4].

In this context, the approach shown in this paper does not present significant number of redundancies, since its implementation addresses only one method of the entire system. Therefore, this approach does not explore the full potential of the Notification-Oriented Paradigm (NOP).

Furthermore, some optimization of NOP implementation may provide better results than the current results, namely in terms of performance. Certainly, these optimizations are related to the development of a particular compiler to solve some drawbacks of the actual implementation of NOP, such as the overhead of using computationally expensive data-structure over an intermediary language. These advances are under consideration in other works.

7. Acknowledgements

R. F. Banaszewski's M.Sc. thesis [10] was supported by CAPES Foundation (Brazil), as well as R. F. Banaszewski's Ph.D. thesis and A. F. Ronszcka's M.Sc. thesis are under CAPES support.

REFERENCES

[1] J. M. Simão, J. M. Simão, R. F. Banaszewski, C. A. Tacla and P. C. Stadzisz, "Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study," *Journal of Software Engineering and Applications*, Vol. 5 No. 6, 2012, pp. 402-416. doi:10.4236/jsea.2012.56047

[2] R. W. Keyes, "The Technical Impact of Moore's Law," *IEEE Solid-State Circuits Society Newsletter*, Vol. 20, No. 3, 2006, pp. 25-27.

[3] E. S. Raymond, "The Art of UNIX Programming," Addison-Wesley, Boston, 2003.

[4] J. M. Simão and P. C. Stadzisz, "Inference Based on Notifications: A Holonic Meta-Model Applied to Control Issues," *IEEE Transactions on Systems*, *Man and Cybernetics*, *Part A*: *Systems and Humans*, Vol. 9, No. 1, 2009, pp. 238-250.

[5] W. Wolf, "High-Performance Embedded Computing: Architectures, Applications, and Methodologies," Morgan Kaufmann, Burlington, 2007.

[6] S. Oliveira and D. Stewart, "Writing Scientific Software: A Guided to Good Style," Cambridge University Press, Cambridge, 2006. doi:10.1017/CBO9780511617973

[7] C. Hughes and T. Hughes, "Parallel and Distributed Programming Using C++," Addison-Wesley, Boston, 2003.

[8] J. M. Simão and P. C. Stadzisz, "Notification Oriented Paradigm (NOP)—A Notification Oriented Technique to Software Composition and Execution," Patent Pending Submitted to INPI/Brazil in 2008 and UTFPR Innovation Agency, 2007.

[9] R. F. Banaszewski, P. C. Stadzisz, C. A. Tacla and J. M Simão, "Notification Oriented Paradigm (NOP): A Software Development Approach Based on Artificial Intelligence Concepts," *VI Congress of Logic Applied to the Technology—LAPTEC*, Santos, 21-23 November 2007, p. 216.

[10] R. F. Banaszewski, "Notification Oriented Paradigm: Advances and Comparisons," Master's Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Techn ology—Paraná (UTFPR). Curitiba, 2009. http://arquivos.cpgei.ct.utfpr.edu.br/Ano_2009/dissertacoes/Dissertacao_500_2009.pdf

[11] M. Herlihy and N. Shavit, "The Art of Multiprocessor Programming," Morgan Kaufmann, Burlington, 2008.

[12] D. Harel, H. Lacover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtulltrauting and M. Trakhtenbrot, "Statemate: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, 1990, pp. 403-416. doi:10.1109/32.54292

[13] D. Sevilla, J. M. Garcia and A. Gómez. "Using AOP to Automatically Provide Distribution, Fault Tolerance, and Load Balancing to the CORBA-LC Component Model," In: C. Bischof, M. Bucker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr and F. Peters, Eds., *Parallel Computing*: *Architectures*, *Algorithms and Applications*, *NIC Series*, John von Neumann Institute for Computing, Jülich, Vol. 38, 2007, pp. 347-354.

[14] W. M. Johnston, J. R. P. Hanna and R. J. Millar, "Advance in Dataflow Programming Languages," *Journal*

ACM Computing Surveys, Vol. 36, No. 1, pp. 1-34, 2004. doi:10.1145/1013208.1013209

[15] G. Coulouris, J. Dollimore and T. Kindberg, "Distributed Systems—Concepts and Designs," Addison-Wesley, Boston, 2001.

[16] W. A. Gruver, "Distributed Intelligence Systems: A New Paradigm for System Integration," *Proceedings of the IEEE International Conference on Information Reuse and Integration*, Las Vegas, 13-15 August 2007, pp. 14-15. doi:10.1109/IRI.2007.4296581

[17] J.-L. Gaudiot and A. Sohn, "Data-Driven Parallel Production Systems," *IEEE Transactions on Software Engineering*, Vol. 16. No. 3, 1990, pp. 281-293. doi:10.1109/32.48936

[18] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm and A. Lain, "The Paradigm Compiler for Distributed Memory Multicomputer," *IEEE Computer*, Vol. 28, No. 10, 1995, pp. 37-47. doi:10.1109/2.467577

[19] P. V. Roy and S. Haridi, "Concepts, Techniques, and Models of Computer Programming," MIT Press, Cambridge, Massachusetts, 2004.

[20] S. Kaisler, "Software Paradigm, Wiley-Interscience," 1st Edition, John Wiley & Sons, New York, 2005.

[21] M. Gabbrielli and S. Martini, "Programming Languages: Principles and Paradigms. Series: Undergraduate Topics in Computer Science," 1st Edition, Springer/Dordrecht Heidelberg, London/New York, 2010.

[22] J. G. Brookshear, "Computer Science: An Overview," Addison-Wesley, Boston, 2006.

[23] A. M. K. Cheng and J.-R. Chen, "Response Time Analysis of OPS5 Production Systems," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, No. 3, 2000, pp. 391-409. doi:10.1109/69.846292

[24] J. A. Kang and A. M. K. Cheng, "Shortening Matching Time in OPS5 Production Systems," *IEEE Transaction on Software Engineering*, Vol. 30, No. 7, 2004, pp. 448-457. doi:10.1109/TSE.2004.32

[25] C. L. Forgy, "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, Vol. 19, No. 1, 1982, pp. 17-37. doi:10.1016/0004-3702(82)90020-0

[26] P.-Y. Lee and A. M. Cheng, "HAL: A Faster Match Algorithm," *IEEE Transaction on Knowledge and Data Engineering*, Vol. 14, No. 5, 2002, pp. 1047-1058. doi:10.1109/TKDE.2002.1033773

[27] M. L. Scott, "Programming Language Pragmatics," 2nd Edition, Morgan Kaufmann Publishers Inc., San Francisco, 2000.

[28] J. M. Simão, "A Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control," Ph.D. Thesis, Federal University of Technology, Paraná; Henri Poincaré University, Nancy, 2005. http://tel.archives-ouvertes.fr/docs/00/08/30/42/PDF/ThesisJeanMSimaoBrazil.pdf

[29] J. M. Simão, C. A. Tacla and P. C. Stadzisz, "Holonic Control Meta-Model," *IEEE Transaction on System, Man & Cybernetics, Part A*, Vol. 39, No. 5, 2009, pp. 1126-1139.

[30] L. V. B. Wiecheteck, "Software Design Method using Notification Oriented Paradigm—NOP," Master's Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology—Paraná (UTFPR), Curitiba, 2011.

[31] R. R. Linhares, A. F. Ronszcka, G. Z. Valença, M. V. Batista, C. R. Erig Lima, F. A. Witt, P. C. Stadzisz and J. M. Simão, "Comparison between Object Oriented Paradigm and Notification Oriented Paradigm under the Context of Telephonic System Simulator," *Internacional Congress of Computation and Telecomunications*, Lima, October 2011.

[32] M. V. Batista, R. F. Banaszewski, A. F. Ronszcka, G. Z. Valença, R. R. Linhares, P. C. Stadzisz, C. A. Tacla and J. M. Simão, "A Comparison between Notification Oriented Paradigm (NOP) and Object Oriented Paradigm (OOP) Carried out by Means of the Implementation of a Sale System," *Internacional Congress of Computation and Telecomunications*, Lima, October 2011.

[33] A. F. Ronszcka, D. L. Belmonte, G. Z. Valença, M. V. Batista, R. R. Linhares, C. A. Tacla, P. C. Stadzisz and J. M. Simão, "Qualitative and Quantitative Comparisons between Object Oriented Paradigm and Notification Oriented Paradigm Based on Play Simulator," *International Congress of Computation and Telecomunications*, Lima, October 2011.

[34] G. Z. Valença, R. F. Banaszewski, A. F. Ronszcka, M. V. Batista, R. R. Linhares, J. A. Fabro, P. C. Stadzisz and J. M. Simão, "NOP Framework, Advances and Comparisons," *Symposium of Applied Computing*, Passo Fundo, 2011.

[35] L. V. B. Wiecheteck, P. C. Stadzisz and J. M. Simão, "A UML Profile to the Notification Oriented Paradigm (NOP)," *Internacional Congress of Computation and Telecomunications*, Lima, October 2011.

[36] B. D. Wachter, T. Massart and C. Meuter, "dSL: An Environment with Automatic Code Distribution for Industrial Control Systems," *Proceedings of the 7th International Conference on Principles of Distributed Systems*, La Martinique, 10-13 December 2003, pp. 132- 145.

[37] A. R. Pimentel and P. C. Stadzisz, "Application of the Independence Axiom on the Design of Object-Oriented Software Using the Axiomatic Design Theory," *Journal of Integrated Design & Process Science*, Vol. 10, No. 1, 2006, pp. 57-69.

[38] S. Ahmed, "CORBA Programming Unleashed," Sams Publisher, Indianapolis, 1998.

[39] D. Reilly and M. Reilly, "Java Network Programming and Distributed Computing," Addison-Wesley, Boston, 2002.

[40] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning," *Proceeding of the 16th European Conference on Object-Oriented Programming*, Springer-Verlag, London, 2002. pp. 178- 204.

[41] M. Díaz, D. Garrido, S. Romero, B. Rubio, E. Soler and J. M. Troya, "A Component-Based Nuclear Power Plant Simulator Kernel: Research Articles," *Concurrency and*

Computation: *Practice and Experience*, Vol. 19, No. 5, 2007, pp. 593-607. doi:10.1002/cpe.1075

[42] S. M. Deen, "Agent-Based Manufacturing: Advances in the Holonic Approach," Springer, Berlin, 2003.

[43] H. Tianfield, "A New Framework of Holonic Self-Organization for Multi-Agent Systems," *IEEE International Conference on Systems*, *Man and Cybernetics*, Montreal, 7-10 October, 2007.

[44] V. Kumar, N. Leonard and A. S. Morse, "Cooperative Control," Springer-Verlag, New York, 2005.

[45] S. Loke, "Context-Aware Pervasive Systems: Architectures for a New Breed of Applications," 1st Edition, Auerbach Publications, Boca Raton, 2006.

[46] A. S. Tanenbaum, M. van Steen, "Distributed Systems: Principles and Paradigms," Prentice Hall, Upper Saddle River, 2002.

[47] I. Sommerville, "Software Engineering," 8th Edition, Addison-Wesley, Boston, 2004.

[48] C. E. Barros Paes and C. M. Hirata, "RUP Extension for the Software Performance," 32*nd Annual IEEE International Computer Software and Applications*, Turku, 28 July 2008, pp. 732-738

[49] G. R Watson, C. E. Rasmussen and B. R. Tibbitts, "An Integrated Approach to Improving the Parallel Application Development Process," *IEEE International Symposium on Parallel & Distributed Processing*, Rome, 23-29 May 2009, pp. 1-8.

[50] J. Giarratano and G. Riley, "Expert Systems: Principles and Practice," PWS Publishing, Boston, 1993.

[51] S. Russel and P. Norvig, "Artificial Intelligence: A modern Approach," Prentice-Hall, Englewood Cliffs, 2003.

[52] D. P. Miranker, "TREAT: A Better Match Algorithm for AI Production System," *Sixth National Conference on Artificial Intelligence—AAAI*'87, Seattle, 13-17 July 1987, pp. 42-47.

[53] D. P. Miranker and B. Lofaso, "The Organization and Performance of a Treat-Based Production System Compiler," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, No. 1, 1991, pp. 3-10. doi:10.1109/69.75882

[54] D. P. Miranker, D. A. Brant, B. Lofaso and D. Gadbois, "On the Performance of Lazy Matching in Production System," 8*th National Conference on Artificial Intelligence AAAI*, Boston, 29 July-3 August 1990, pp. 685-692.

[55] D. Watt, "Programming Language Design Concepts," J. W. & Sons, Baltimore, 2004.

[56] T. Faison, "Event-Based Programming: Taking Events to the Limit," Apress, New York, 2006.

[57] S. M. Tuttle and C. F. Eick, "Suggesting Causes of Faults in Data-Driven Rule-Based Systems," *Proceedings of the IEEE* 4*th International Conference on Tools with Artificial Intelligence*, Arlington, 10-13 November 1992, pp. 413-416. doi:10.1109/TAI.1992.246438

