Notification Oriented Paradigm (NOP) and Object Oriented Paradigm (OOP): A Comparative Study by means of a Sale Order System

Cleverson Avelino Ferreira[3], Dr. Jean M. Simão[1], Dr. Paulo Cézar Stadzisz[1], Márcio V. Batista[2]

cleversonavelino@gmail.com, jeansimao@utfpr.edu.br, stadzisz@utfpr.edu.br, marcio.venancio@gmail.com

[1] Professors at Graduate School in Electrical Eng. & Industrial Computer Science (CPGEI) and Graduate School of Applied Computing (PPGCA). [2] Masters Degree Student at CPGEI/UTFPR. [3] Masters Degree Student at PPGCA/UTFPR.
Federal University of Technology, Brazil
Av. 7 de Setembro, 3165
Curitiba - Brazil

Abstract: *This paper presents a comparative study between Notification Oriented Paradigm (NOP) and Object Oriented Paradigm (OOP) by means as an experiment. The OOP has problems which can lead the developers to build systems with low quality. These problems are related to unnecessary casual expressions evaluation (i.e. if-then statements or similar) and higher software entity coupling. In this context, Notification-Oriented Paradigm (NOP) presents an alternative for those issues. NOP proposes another way to structure software and make its inference, which is based on small, collaborative, and decoupled computational entities whose interaction happens through precise notifications. This paper presents a quantitative comparison, time evaluation performance, between two equivalent versions of a sale system, one developed according to the principles of OOP in C++ and another developed according to the principles of NOP based on a current NOP framework over C++. The experiment results have shown that OOP version has obtained better runtime perform-ance than NOP implementation. This happened because the NOP framework uses considerable expensive data-structures over C++. Thus, it is necessary a real compiler to NOP or at least a highly optimized NOP framework in order to use its potentiality indeed. Besides, in a scenario variation of approvable causal expressions, the experiment results have shown an increase in the number of causal expression unnecessary evaluated in OOP version, whereas the NOP version does not present unnecessary causal expression evaluation. Indeed, by definition, NOP application does not waste execution time unnecessarily evaluating causal expressions*.

Keywords: Notification Oriented Paradigm, Notification Oriented Inference, NOP and IP Comparison.

1 Introduction

This section mentions drawbacks from current programming paradigms, introduces Notification Oriented Paradigm (NOP) as a new solution, and presents 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]. 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 [2].

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 [3]. Such systems normally employ less-powerful processors due to factors such as constraints on power consumption and system price to a given market [4].

Besides, computational power misusing in software can also cause overuse of a given standard processor, implying in execution delays [3][5]. Still, in complex software, this can even exhaust a processor capacity, demanding faster processor or even some sort of distributions (e.g. dual-core) [3][6]. Indeed, an optimization-oriented programming could avoid such drawbacks and related costs [3][7].

Therefore, suitable engineering tools for software development, namely programming languages and their environments, should facilitate the development of optimized and correct code [8][9][10][11]. Otherwise, engineering costs to produce optimized-code could exceed those of upgrading the processing capacity [3][8][9][10].

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 [14][15][16][17]. However, the 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 [3][14][15][18].

In this context, a problem raises from the fact that 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 [2][3][18][19]. This happens due to the structure and execution nature imposed by their paradigm [6][8][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 [9][20][21]. 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 [9][20][22].

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 create programs as a monolithic entity comprising prolix and coupled code, generating non-optimized and interdependent code execution [7][8] [22][23].

Declarative Paradigm is the alternative to the Imperative Paradigm. Essentially, it enables a higher level of abstraction and easier programming [21][22]. 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 [24][25][26][27]. 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 [7][8].

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 Paradigm solutions [9][28]. Furthermore, similarly to the Imperative Paradigm programming, the Declarative Paradigm programming also generates code coupling due to the similar search-based inference process [3][7][22].

Still, other approaches between them, such as event-driven and functional programming, do not solve these problems even if they may reduce some problems, like reduce certain redundancies [23][28]. Actually, all these issues have been minutely taken into account in previous works, e.g. [3][7][8][9][9].

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 [3][7][8]. Therefore, this impels 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 in order to solve some of the highlighted problems [3][7][8].

The NOP embryonic basis was initially proposed by J. M. Simão as a manufacturing discrete-control solution [12][29]. This solution was evolved as general discrete-control solution and then as a new inference-engine solution [3], achieving finally the form of a new programming paradigm [7][8][9].

The essence of NOP is its inference process based on small, smart, and decoupled collaborative entities that interact by means of precise notifications [3]. This solves redundancies and centralization problems of the current causal-logical processing, thereby solving processing misuse and coupling issues of current paradigms [3][7][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 this paper is evaluated the current NOP materialization in terms of performance, which is available to use as a result of a M. Sc. Thesis [51].

2 Notification Oriented Paradigm (NOP)

The Notification Oriented Paradigm (NOP) introduces a new concept to conceive, construct, and execute software applications. 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 [3] [7]. This allows enhancing software applications performance and potentially makes easier to compose software, both non-distributed and distributed ones [9].

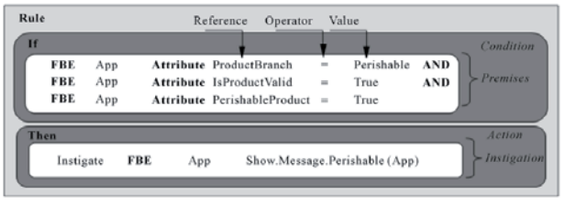

Figure 1. Rule Entity.

2.1 NOP Structural View

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

Structurally, a Rule has two parts, namely a “Condition” and an “Action”, as shown by means of the UML class diagram in Figure 2. 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 [8].

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” [8]. Attributes states are evaluated in the Conditions of Rules by associated entities called “Premisses”. In the example, which is shown by the

figure 1, the Condition of the Rule is associated to three Premises, which verify the FBE Attributes state as follow: 1) Is the product branch perishable? 2) Is the product valid? 3) Is the product perishable date? [8].

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 [8].

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 [8].

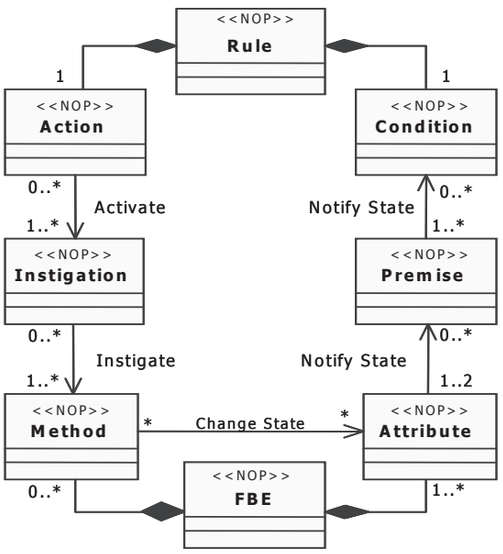

Figure 2. Rule and Fact_Base_Element class diagram.

2.2 NOP Inference Process

The inference process of NOP is innovative once the Rules have their inference carried out by active collaboration of its notifier entities [3]. 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 by means of punctual notifications between the collaborators.

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: (a) a reference to an Attribute discrete value, called *Reference*, which is received by notification; (b) a logical operator, called *Operator*, useful to make comparisons; and (c) another value called *Value* that can be a constant or even a discrete value of other referenced Attribute.

A Premise makes a logical calculation when it receives notification of one or even two Attributes (i.e. *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.

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.

The collaboration between NOP entities by means of notifications can be observed at the schema illustrated in Figure 3. In this schema, the flow of notifications is represented by arrows linked to rectangles that symbolize NOP entities.

An important point to clarify about NOP collaborative entities is that each notifier one (e.g. Attributes) registers its client ones (e.g. Premises) in their creation. 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.

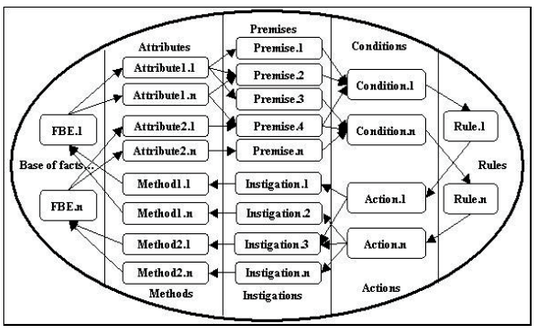

Figure 3. Notification chain of Rules and collaborators [3]

2.3 NOP Implementation

In order to provide the use of these solutions before the conception of a particular language and compiler, the NOP entities were materialized in C++ programming language in the form of a framework and the applications developed have been made just by instantiating this framework [9]. 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.

3 The Sale Order System

In order to do a comparison between Notification Oriented Paradigm (NOP) and Oriented Object Paradigm (OOP), a Sales Order System was created. This system was used as a case of study with proposal to observe the elapsed time in two test scenarios.

The OO version was built over C++ language and the NOP version runs with NOP framework. This framework was created over C++.

3.1 Requirements

The proposed software system, thought to lead the comparison between NOP and OOP versions, has the following functional requirements and non-functional requirements:

FR1	The system shall allow selling products.

Table 1. Functional Requirements.

SFR1.1	The system do not allow selling products with stock equals zero
SFR1.2	The system shall allow sending more than one product per sale order
SFR1.3	The system shall persist the sale order information
SFR1.4	The system has to calculate the total price of sale regarding customer classification.

Table 2. Sub - Functional Requirements for Functional Requirement 1.

NFR1	To be implemented in two versions, OOP in C++ and NOP Framework over C++.

Table 3. Non – Functional Requirements.

3.2 Sale Order System – Structure

In order to build the Sale Order System a Class Diagram was created. This diagram shows the components that had to be developed. Fig. 4 shows the Sales Order System class diagram.

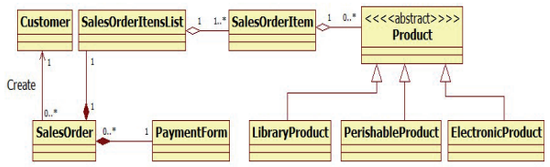

Figure 4. Sale Order System (NOP and OOP) class diagram

As seen in Fig. 4 the most important class is the SalesOrder. This class has an association with Customer, PaymentForm and SalesOrderItensList classes. The SalesOrderItensList has been used to support an association with SalesOrderItem which is used to maintain the information about the selected products.

3.3 Sale Order System – Execution

The sale starts with the customer code that passes through verification if the customer has the system access. After that, the customer has to choose the payment form type. There are just two payment ways available, which are in cash and installment payment.

The Sale process continues asking to the customer what products he wants to buy and checks if the product is available in the stock. After, the System has to calculate the discount price for the product. On the customer profile has a parameter which is used to inform the classification type of the customer. This classification is used to provide a special form of discount to certain customers. There is a sort of customer classification types that allows discount from 5% up to 95%.

After the whole cycle of product insertion in the sales order, the sale can be closed. If the customer chooses installment payment form, the system has to check if the customer has available credit limit to buy the desired items. Actually, the system has the information about the credit limit of the customers.

3.4 Implementation details

The first system version made was in OOP. The fig. 5 shows the code in OOP with causal expressions used to give the discount percentage for a Customer Sales Order. This piece of code was chosen because describes the most important part from the discount calc process. Each *if* statement is responsible to evaluate the customer type. Once the statement results a true condition the discount percentage is returned by the method. There are many customer types and for each evaluation many *if* statements are evaluated unnecessarily which shows a waste of processing time.

```
79  double SalesOrder::getCalculoDiscount() {
80      double discount = 0.0;
81      this->somaIF();
82      if (this->getCustomer()->getCustomerType() == 1) {
83          discount = 5;
84          return discount;
85      }
86
87      this->somaIF();
88      if (this->getCustomer()->getCustomerType() == 2) {
89          discount = 10;
90          return discount;
91      }
92
93      this->somaIF();
94      if (this->getCustomer()->getCustomerType() == 3) {
95          discount = 15;
96          return discount;
97      }
98
99      this->somaIF();
100     if (this->getCustomer()->getCustomerType() == 5) {
101         discount = 25;
102         return discount;
103     }
104
105     ...
106
```

Figure 5. OOP code for discount type.

Once the development of OOP version was finished, the development of NOP version was taken into account. In this version the same patterns were used. However, of-course, there are differences between OOP and NOP implementations. Fig. 5 and Fig. 6 show the respective differences between the two implementations to give a discount price by the customer type.

```
1   INTEGER(this, atTypeDiscount, -1);
2
3   for (int i = 1; i <= 20; i++) {
4       RULE(rlDiscountType, scheduler, Condition::SINGLE);
5       rlDiscountType->
6           addPremise(atTypeDiscount, i, Premise::EQUAL,Premise::STANDARD , false);
7       rlDiscountType->
8           addMethod(this, atPercDescount, i * 5, Attribute::STANDARD);
9   }
```

Figure 6. NOP code for discount type.

The difference between the applications could be observed by means of causal expressions. There are no more if-then causal tests and nested code in NOP version. The entire sale flow is governed by *Rules*, *Conditions*, *Premises*, *Attributes*, and other collaborator smart entities. Every time which an *Attribute* has its state changed, it starts a notification process.

The fig. 10 particularly shows the code used in NOP application. This piece of code was written in the SalesOrder class constructor. There were configured in the SalesOrder class two *Attributes* in order to return the discount percentage for the Sales Order.

The first *Attribute* is called *atTypeDiscount* which is used to configure the customer type value. The second *Attribute* is called *atPercDiscount* which is used by the System to get the percentage discount value.

When the System starts, the *atTypeDiscount Attribute* is configured with the default value. This configuration is shown in the first line of the code.

The second line shows a *for* statement which is used to create the *Rules* that have to control the type of discount given.

The *atPercDiscount Attribute* is used by the System when it is necessary to know the percentage discount for a specific Product that the Customer is adding into his Sales Order.

As known there are twenty Customer type possibilities and every loop iteration allows creating one *Rule* which is responsible for evaluating a specific Customer type and determines the percentage discount. Inside each *Rule* is configured a *Premise* and a *Method*.

In the fourth line is configured a *Premise* which is responsible to receive the *atTypeDiscount Attribute* that is used by the notification process. The value used to do this comparison is the second parameter, the variable *i,* that in each loop iteration is configured with an increment by one. The third parameter shows the expected behavior when the *atTypeDiscount Attribute* has its value changed. The system has to evaluate a *Premise* comparing if the *atTypeDiscount* is equal than the *i* value.

In turn, in the fifth line it was added a *Method* which is responsible to configure the discount value at the *atPercDescount Attribute*. According with the NOP structure it is necessary, at this point, create an *Instigation Object* that is used by the inference process to call the related method, but with the improvements achieved by the NOP framework [51] when it is created a new *Method* the framework will create all the necessary objects to make the *Rule* works correctly, in this case the necessary objects are *Actions* and *Instigations Objects*. The third parameter in this method is used to get the percentage value and set at the *atPercDescount* when this method is invoked by the notification process.

The notification process to configure the discount percentage starts when the *atTypeDiscount Attribute* receives a new Customer type value. The new Customer type value will be configured at the *atTypeDiscount Attribute* every time that the Customer is registered at a new Sales Order. Thereafter, all related *Premisses* will be evaluated in order to compare its *Conditions* with the new *atTypediscount* value.

When a specific *Condition* results in a true state the *Rules* becomes true and can activate its *Actions* that are composed of special-entities called *Instigations*. The *Instigation*, in this case, will instigate the execution of a related *Method* to set the configured value at the *atPercDescount*.

4 A Performance Study

This section presents a performance study between the OOP and NOP Sales Order versions. The objective of this study is to show the tests performance evaluation about these two versions.

In order to provide the evaluation, an amount of Sales Orders were executed from each version. The main goal was to verify how long time each version takes to finish the process. There were made two types of experimentation which will be discussed in the next sections.

4.1 First Experiment and results

The experiments are performed using the optimized version of NOP. In this version the NOP Framework structure, which includes the notification chain, was changed with some optimizations and refactoring with respect to two previous versions. According with [51] these improvements have achieved 50% of performance time gain with respect to the last preview version.

Once this considered, the expectation of this first experiment was to verify the time performance in the current Sale Order System of its implementation in the current NOP framework against its implementation in the OOP.

The first experiment was designed to evaluate the performance of each implementation/application in order to create 100, 1000, and 10000 combinations of predetermined sales order. Each customer, product, and payment form will generate one sales order combination.

At the table 4 it is possible to observe the data used in this first experiment. These data present four Customers which start with identification 1 to 4 and type 1 to 4. Also, there are two Products with identification 1 and 2 and two available payment forms.

The configuration to execute this first experiment has been developed to get low causal expression evaluation by means the value type which was configured in each *Customer*. For each *Customer* type, the System will calculate a discount price for the Product and for each Customer type in the OOP version it is necessary to verify this information though a set causal expressions (“ifs”) where frequently most of them are unnecessarily evaluated.

For example, if the customer is from the type 19, the system will evaluate 18 unnecessary causal expressions until get the right decision. In the NOP application, in turn, the notifiable *Rules* and its collaborators are responsible to manage this behavior.

Customer	
	Customer #1 Type #1
	Customer #2 Type #2
	Customer #3 Type #3
	Customer #4 Type #4
Product	
	Product #1
	Product #2
Payment Form	
	Cash #1
	Installment #2

Table 4. First experiment content.

This experiment has been made using an Operational System (OS) Windows 7 and a computer Intel processor with 1.30 Giga hertz and 2 Giga bytes of RAM memory.

The table 5 shows the performance evaluation of the Sales Order System. The second column shows the Elapsed time that OOP took to perform the correspondent amount of sales. The third column shows the number of causal expression in OOP which are used to manage the expected result. Finally, the fourth column shows the elapsed time from NOP version to create the correspondent amount of Sales Order.

Still, this table shows that OOP version had a less execution time when compared with the NOP version. But the number of causal expressions evaluated with OOP version in this first experiment was extremely large. In turn, in the NOP version due to the use of its *Rules* and its smart collaborators *Entities* the number of causal expression evaluated is irrelevant or equals one.

In this case of study is important to say that the NOP version uses the current NOP Framework built over the C++ programming language which can cause some drawbacks, such as the overhead of using computationally expensive data-structure over an intermediary language.

Total of Sales Order Combination	OOP (Milliseconds)	Number of Causal Expressions (OOP)	NOP (Milliseconds)
100	0,0036	1440	0,0075
1000	0,0111	15000	0,0210
10000	0,1671	137440	0,2393

Table 5. First experiments results.

4.2 Second Experiment and results

In this second experiment the number of causal expression evaluation has increased. As seen in the table 6, the customer configuration type has been changed to the worse case. Thus, the customer type has changed to the last options which are evaluated by the causal expressions. In this situation the OOP application has to evaluate a great amount of causal expressions which cause a processor waste of time.

Customer	
	Customer #16 Type #16
	Customer #17 Type #17
	Customer #18 Type #18
	Customer #19 Type #19
Product	
	Product #1
	Product #2
Payment Form	
	Cash #1
	Installment #2

Table 6. Second experiment content.

The table 7 shows the results of the second experiment between both versions. Again, this second experiment has shown a less execution time from OOP when compared with the NOP version to run the experiment. As shown in the table 7 the number of causal expression evaluated has increased as well.

Total of Sales Order Combination	OOP (Milliseconds)	Number of Causal Expressions (OOP)	NOP (Milliseconds)
100	0,0037	3456	0,0076
1000	0,0118	36000	0,0211
10000	0,1882	347440	0,2395

Table 7. Second experiments results.

4.3 Additional comparison

This section will discuss the elapsed execution time difference between both versions. It is necessary to show how important is to observe this situation. Here, both table 8 and table 9 shows a different scenario of comparison. In these scenarios is evaluated the performance between the first and second experiment for each application version, i.e. NOP and OOP versions.

As seen in table 8, the first column shows the total amount of sales order created. The second column shows the elapsed execution time taken from the first experiment. The third column shows the elapsed execution time taken by the second experiment and the fourth column shows the percentage difference between the execution time from both versions.

In the NOP version, which is show in the table 8, it is possible to observe that the elapsed execution time between the first and second experiment did not considerably increase. It was because the type of evaluation done using NOP application which uses smart entities and *Rules* to manage the expected behavior.

In the OOP version, which is show in the Table 9, it is possible to observe an execution time increasing between the first and second experiments. As seen before, the number of causal expression evaluated within both scenarios was extremely different. In the first experiment these number of expressions were lower than the second experiment which describes why the first experiment has less execution time when compared with the second experiment.

Total of Sales Order Combination	PON(1)	PON(2)	Difference (%)
100	0,0075	0,0076	1.33%
1000	0,021	0,0211	1.48%
10000	0,2393	0,2395	1.08%

Table 8. NOP - Increase of time because of causal expressions.

Total of Sales Order Combination	OOP(1)	OOP(2)	Difference (%)
100	0,0036	0,0037	2.78%
1000	0,0111	0,0118	6.31%
10000	0,1671	0,1882	12.63%

Table 9. OOP - Increase of time because of causal expressions.

Through this comparison it is possible to observe the behavior differences between both versions by means the causal expression evaluation. The OOP version has shown an execution time increase in the second experiment compared with the first experiment in order to create the same amount of sales order. In turn, the NOP version has presented that the execution time from both versions were almost the same which shows that NOP version has not wasted the execution time to evaluating causal expression unnecessarily.

5 Conclusion and Future Works

This section discusses NOP properties and NOP Performance.

5.1 NOP 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 is 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][30].

The notification mechanism is composed of entities that collaboratively carry out the inference process by means of notifications, 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.

5.2 NOP Performance

As demonstrated in this paper, NOP could decrease the loss of processing time to evaluate causal expressions unnecessary by means of its innovative notification mechanism [3][7].

This mechanism assures that each change of "variable" (i.e. FBE Attribute) state activates only the strictly necessary evaluations of logical and causal expressions (i.e. Premises and Conditions of Rules) [3][9]. It was possible to see this behavior with the presented comparisons in this paper.

Also, NOP would improve the performance by sharing the results of logic evaluation (i.e. notification of Premises) between causal evaluations (i.e. execution of Conditions), therefore avoiding unnecessary repetitions of code and processing in the execution of the Rules [3]. Thus, temporal and structural redundancies are avoided by NOP, theoretically guarantying suitable performance by definition [3]. Still, NOP has been analyzed through an Asymptotic Analysis of the Complexity. The result of this analysis has shown that NOP implies an O(n) complexity which is an excellent result [12].

Furthermore, some optimization of NOP implementation may provide better results than the current results, namely in terms of runtime 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.

References

[1] R.W. Keyes. "The Technical Impact of Moore's Law". IEEE solid-state circuits society newsletter. IBM T. J. Watson Research Center, 2006.

[2] E.S. Raymond, "The Art of UNIX Programming", pp.327, A.Wesley, 2003.

[3] J. M. Simão, P. C. Stadzisz, "Inference Based on Notifications: A Holonic Meta-Model Applied to Control Issues". IEEE Transaction on System., Man, and Cybernetics, Part A V. 9 Issue 1 Pg 238-250, 2009. Doi. 10.1109/TSMCA.2008.2006371.

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

[5] S. Oliveira, D. Stewart, "Writing Scientific Software: A Guided to Good Style". Cambridge Univ. Press, 2006.

[6] C. Hughes and T. Hughes. Parallel and Distributed Programming Using C++. Addison Wesley, 2003.

[7] J. M. Simão, P. C. Stadzisz, "Paradigma Orientado a Notificações (PON) - Uma Técnica de Composição e Execução de Software Orientada a Notificações". Patent pending submitted to INPI/Brazil in 2008 and UTFPR Innovation Agency 2007. N.: PI0805518-1.

[8] R. F. Banaszewski, P.C. Stadzisz, C.A. Tacla, J. M Simão, "Notification Oriented Paradigm (NOP): A Software Development Approach based on Artificial Intelligence Concepts". VI Congress of Logic Applied to the Technology, Paper 216, Santos, Brazil, 2007.

[9] R.F. Banaszewski, "Paradigma Orientado a Notificações: Avanços e Comparações". M.Sc. Thesis, CPGEI/UTFPR. Curitiba-PR, 2009.

[10] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.

[11] D. Harel, H. Lacover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtulltrauting, M. Trakhtenbrot. "Statemate: a working environment for the development of complex reactive systems" IEEE Transaction on Software Engineering. V. 16, n. 4, pp. 403-416, 1990.

[12] 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 CPGEI/UTFPR/Brazil & CRAN/UHP/France, 2005. http://tel.archives-ouvertes.fr/docs/00/08/30/42/PDF/ThesisJeanMSimaoBrazil.pdf.

[13] B. D. Wachter, T. Massart, C. Meuter. "dSL: An Environment with Automatic Code Distribution for Industrial Control Systems", Proc. of the 7th Int. Conf. on Principles of Distributed Syst., 2003, La Martinique, France, V. 3144 of LNCS, pg 132-45, Springer, 2004.

[14] D. Sevilla, J.M. Garcia, A. Gómez. "Using AOP to Automatically Provide Distribution, Fault Tolerance, and Load Balancing to the CORBA-LC Component Model". Parallel Computing: Architectures, Algorithms, and Applications, J. von Neumann Institute for Computing, Jülich, NIC Series, V.38, 2007. Reprinted: Advances in Parallel Computing, V. 15, 2008.

[15] W. M. Johnston, J. R. P. Hanna, R. J. Millar, "Advance in Dataflow Programming Languages". Journal ACM Computing Surveys, Volume 36, No. 1, pp. 1-34, march 2004.

[16] G. Coulouris, J. Dollimore, T. Kindberg, "Distributed Systems – Concepts and Designs". Reading, MA: Addison-Wesley, 2001.

[17] W. A. Gruver, "Distributed Intelligence Systems: A new Paradigm for System Integration". Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI), pg 14-15, 2007.

[18] J-L Gaudiot, A. Sohn, "Data-Driven Parallel Production Systems". IEEE Transaction on Software Engineering. V. 16. No 3, pg 281-293, 1990.

[19] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, "The Paradigm Compiler for Distributed Memory Multicomputer". IEEE Computer, 28 (10), pp. 37-47, 1995.

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

[21] S. Kaisler, "Software Paradigm", Wiley-Interscience, 1st Edition, 0471483478 John Wiley & Sons, 2005.

[22] M. Gabbrielli, S. Martini, "Programming Languages: Principles and Paradigms". Series: Undergraduate Topics in Computer Science. 1st Edition, 2010, XIX, 440 p., Softcover. ISBN: 978-1-84882-913-8.

[23] J. G. Brookshear. "Computer Science: An Overview". Ad. Wesley 2006.

[24] A. M. K. Cheng and J-R. Chen. "Response Time Analysis of OPS5 Production Systems". IEEE Transactions on Knowledge and Data Engineering, v. 12, n.3, pp. 391-409, 2000.

[25] J. A. Kang and A. M. K. Cheng. "Shortening Matching Time in OPS5 Production Systems". IEEE Transaction on Software Engineering. V. 30, N. 7, 2004.

[26] C. L. Forgy, "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence N. 19, pg 17-37, 1982.

[27] P.-Y. Lee, A. M. Cheng, "HAL: A Faster Match Algorithm". IEEE Transaction on Knowledge and Data Engineering, 14 (5), pp. 1047-1058, 2002.

[28] M. L. Scott, "Programming Language Pragmatics", 2º Edition, p. 8, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2000.

[29] J. M. Simão, C. A. Tacla, P. C. Stadzisz, "Holonic Control Meta-Model". IEEE Transaction on System, Man, and Cybernetics, Part A. V. 39, N. 5, 2009. Doi. 10.1109/TSMCA.2009.2022060

[30] A.R. Pimentel; 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, v. 10, 2006.

[31] S. Ahmed. "CORBA Programming Unleashed". Sams Pub. 1998.

[32] D. Reilly, M. Reilly. "Java Network Programming and Distributed Computting". Addison-Wesley, 2002.

[33] E. Tilevich, Y. Smaragdakis. "J-Orchestra: Automatic Java Application Partitioning" 16th European Conf. on Object-Oriented Programming, pg 178-204, B. Magnusson (Ed), Springer, 2002.

[34] S. Loke, "Context-Aware Pervasive Systems: Architectures for a New Breed of Applications". Auerbach Publications, 2006.

[35] M. Díaz, D. Garrido, S. Romero, B. Rubio, E. Soler, J. M. Troya, "A component-based nuclear power plant simulator kernel: Research Articles". Concurrency and Computation: Practice and Experience, 19 (5), pp. 593 - 607, 2007.

[36] S. M. Deen, "Agent-Based Manufacturing: Advances in the Holonic Approach", Springer, 2003, ISBN 3-540-44069-0.

[37] H. Tianfield "A New Framework of Holonic Self-organization for Multi-Agent Systems" IEEE International Conference on System, Man and Cybernetics, 2007.

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

[39] A. S. Tanenbaum, M. van Steen, "Distributed Systems: Principles and Paradigms", (Book) Prentice Hall, 2002.

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

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

[42] D. P. Miranker, "TREAT: A better Match Algorithm for AI Production System" 6th National Conference on AI, pp. 42-47, 1987.

[43] D. P. Miranker, B. Lofaso. “The organization and Performance of a TREAT-based Production System Compiler”. IEEE Transactions on Knowledge and Data Engineering, III (1), pp. 3-10, 1991.

[44] D. P. Miranker, D. A. Brant, B. Lofaso, D. Gadbois. “On the Performance of Lazy Matching in Production System”. 8th National Conference on Artificial Intelligence, pp. 685-692, AIII/The MIT Press, 1992.

[45] D. Watt, “Programming Language Design Concepts”. J. W. & Sons, 2004.

[46] T. Faison. "Event-Based Programming: Taking Events to the Limit”. Apress, 2006.

[47] S. M. Tuttle, C. F. Eick, “Suggesting Causes of Faults in Data-Driven Rule-Based Systems”. Proc. of the IEEE 4th International Conference on Tools with Artificial Intelligence, pg 413-416, Arlington, VA., 1992.

[48] C. E Barros Paes, C. M. Hirata, “RUP Extension for the Software Performance”. 32nd Annual IEEE International Computer Software and Applications (COMPSAC '08), pp 732-738, July 28 2008.

[49] G. R Watson, C. E. Rasmussen, B. R. Tibbitts, “An integrated approach to improving the parallel application development process”. IEEE International Symposium on Parallel & Distributed Processing, pp 1 - 8, 2009.

[50] I. Sommerville, “Software Engineering”, 8th Ed. Ad. Wesley, 2004.

[51] Valença, G. Z. “Contribution to the materialization of Notification Oriented Paradigm (NOP) by framework and wizard.”, M.Sc. Thesis. PPGCA/UTFPR. Curitiba-PR, 2012.