Journal of Software Engineering and Applications, **2012, 5, 402-416**
http://dx.doi.org/10.4236/jsea.2012.56047 Published Online June 2012 (http://www.SciRP.org/journal/jsea)

Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study

Jean M. Simão[1,2], Cesar A. Tacla[1,2], Paulo C. Stadzisz[1,2], Roni F. Banaszewski[1]

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

Received March 22nd, 2012; revised April 19th, 2012; accepted April 28th, 2012

ABSTRACT

This paper presents a new programming paradigm named Notification Oriented Paradigm (NOP) and analyses performance aspects of NOP programs by means of an experiment. NOP provides a new manner to conceive, structure, and execute software, which allows better performance, causal-knowledge organization, and entity decoupling 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 thanks to the concept of high-level programming. However, they are considered slower to execute and lesser flexible to program than IP. Anyway, both paradigms present similar drawbacks like causal-evaluation redundancies and strongly coupled entities, which decrease software performance and processing distribution feasibility. These problems exist due to an orientation to monolithic inference mechanism based upon sequential evaluation by means of searches over passive computational entities. NOP proposes another manner to structure software and make its inferences, which is based upon small, smart, and decoupled collaborative entities whose interaction happen by means of precise notifications. This paper discusses NOP as a paradigm and presents certain comparison of NOP against IP. Actually, performance is evaluated by means of IP and NOP programs with respect to a same application, which allow demonstrating NOP superiority.

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 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 not 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 [8-13]. Otherwise, engineering costs to produce optimized-code could exceed those of upgrading the processing capacity [3,8-10].

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 [14-17]. 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 [24-27]. 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].

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 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 [7-9].

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 [3,7-9].

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 Paradigm.

In short, the study shows NOP advantages to save processing. Moreover, it allows presenting other research perspectives with respect to NOP. For instance, it allows showing the suitability to distribution by highlighting the achieved decoupling degree of NOP (code) elements.

2. Background

This section explores programming paradigm drawbacks.

2.1. Imperative Programming Issues

The main drawbacks of Imperative Programming are concerned to the related code redundancy and coupling [3]. The first mainly affects processing time and the second processing distribution, as detailed in the next subsections.

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 [7,9].

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.


```
...
while (true) do
  if((object_1.attribute_1 = 1) and
     (object_2.attribute_1 = 1) and
     (object_3.attribute_1 = 1))
  then
      object_1.method_1();
      object_2.method_1();
      object_3.method_1();
  end_if
  ...
  if((object_1.attribute_1 = 1) and
     (object_2.attribute_n = n) and
     (object_3.attribute_n = n))
  then
      object_1.method_n();
      object_2.method_n();
      object_3.method_n();
  end_if
end_while
...
```

Figure 1. Example of imperative code.

This type of code causes the problem called, in the computer science, temporal and structural redundancy [3,26].

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 [3]. For instance, the logical expression (*object* 1. *attribute* 1 = 1) is replicated in several causal expressions (*i.e*. *if-then statements*) [3,7].

These redundancies can be seen unimportant in this didactic code example, mainly if the number (*n*) of causal expressions is small. However, even with better code, 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 [7,9].

The code redundancies may result, for example, in the need of a more powerful processor than it is really required [3,6]. 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 (*i.e*. "*if-then*") calculation [3,6,8].

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 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 [3].

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 [3].

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 module, or even the whole program, a monolithic computational unit [3].

2.1.3. Imperative Programming Distribution Hardness

When distribution is intended (e.g. process, processor, and cluster distribution), an analysis of code could identify less dependent code sets to facilitate their splitting. However, this is normally a complex activity due to the code coupling and complexity caused by the imperative programming [13,19].

In this sense, well-designed software composed of modules as decoupled as possible, using advanced and quite complicated software engineering concepts like *aspects* [14] and *axiomatic design* [30], 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 [14,31,32].

In spite of those advances, distribution of single code elements or even code modules is still a complex activity demanding research efforts [13-15,18,33,34]. 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. fine-grained, balanced, and minimal inter-dependent distribution) [3].

Indeed, distribution hardness is an issue because there are contexts where distribution is actually necessary [6, 16,17]. For instance, a given optimized program exceeding the capacity of an available processor would demand processing splitting [5]. Other instances are programs that must guaranty error isolation or even robustness by distributed module redundancy [12]. These features can be found in application of nuclear-plant control [35], intelligent manufacturing [12,29,36,37], and cooperative controls [38].

Besides, there are other applications that are inherently distributed and need flexible distribution, such as those of ubiquitous computing. More precise examples are sen-

sor networks and some intelligent manufacturing control [34,37]. Moreover, the easy and correct distribution is an expectation due to the reduction of processor prices and the communication networks advances as well [9,39].

2.1.4. Imperative Programming Development Hardness

In addition to optimization and distribution issues, the program development with Imperative Programming can be seen as hard due to complicated syntax and a diversity of concepts to be learned, such as pointers, control variables, and nested loops [40]. The development process would be error-prone once a lot of code still comes from a manual elaboration using those concepts. In this context, the exemplified imperative algorithm (**Figure 1**) could be certainly optimized, however without significant easiness in this activity and true fine-grained code decoupling.

It would be necessary to investigate better solutions than those provided by Imperative Paradigm. A solution to solve some of its problems may be the use of programming languages from another paradigm, such as Declarative Programming that automates the evaluation process of causal expressions and concerned elements [20,41].

2.2. Declarative Programming Issues

A well-known example of Declarative Programming and its nature is Rule Based System (RBS) [3,40]. A RBS provides a high-level language in the form of causal-rules, which prevents the developers from algorithm particularities [40]. 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) [41].

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) [25,40]. Moreover, some IE algorithms (e.g. RETE [24-26], TREAT [42,43], LEAPS [44], and HAL [27] algorithms) avoid most of temporal and structural redundancies [9]. However, the data structures used to solve redundancies in those IEs implies in too much consuming of processing capacity [26].

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 [9,23,28,45].

A solution to these problems can be the symbiotic use of Declarative and Imperative Programming [20,45]. Indeed, such approach has been presented, like CLIPS++, ILOG, and Rules. However, they are not popular due to factors such as syntax mixing, paradigms mixing, and technical cultural reasons [9]. Anyway, even Declarative Paradigm being a relevant solution, it does not solve some problems.

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 on a search technique (*i.e.* matching) implies a strong dependency between facts and rules because they together constitute the search space [3].

2.3. Other Programming Approach Drawbacks

Enhancements in the context of Imperative and Declarative Paradigm have been provided to reduce the effects of recurrent loops or searches, such as event-driven programming and functional programming [9,41,46]. Event programming and functional programming have been used to different software such as discrete control, graphical interfaces, and multi-agent systems [9,41,46].

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.

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.

An alternative programming approach is the Data Flow Programming [15] 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 [15]. The distribution in Data Flow Programming is achieved in arithmetical processing, however it is not really achieved in logical-causal calculation [15,18]. This calculation is carried out

by means of current advanced inference engines, namely Rete [18,47].

The fact is that 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.

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:

- Effective code optimization to be sure about the eventual need of a faster processor and/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 considering the increasing market demand by software, where development easiness, code optimization, and processing distribution are current requirements [48-50]. Indeed, this software development "crisis" impels new researches and solutions to make simpler the task of building better software.

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). In addition, NOP evolves some of their concepts and solves some of their deficiencies [3,7,9].

3. Marksmanship Game

Before NOP concepts were firstly used to discrete control applications for quite diversified and complex simulated manufacturing systems. The simulator used was ANALYTICE II, developed at CPGEI/UTFPR. Specifically, concepts of the nowadays called NOP were used to build a control meta-model, which allows instantiating control applications, particularly to ANALYTICE II [12]. Those concepts revealed to be suitable to control applications [12,29].

In a given period of time, the solution was called Holonic Control Meta-Model due to its holistic features and its applicability to the so called Holonic Manufacturing Systems [29]. Nowadays, this Holonic Control Metamodel is also called Notification Oriented Control (NOC). Besides, NOC is considered the genesis of the now called Notification Oriented Inference (NOI). In turn, NOI is considered the genesis of NOP. Thus, discrete control applications of NOC could be interpreted as a NOP domain application.

Nevertheless, each control application over ANALYTICE II is actually complex to be used in a comparison study between NOP and other paradigms, such as comparison between NOP and Imperative Paradigm. Indeed, the understanding of complex application could undermine NOP understanding and the experiment understanding as well. Thus, to better explain differences between these paradigms, another and simpler application is here proposed aiming at the NOP nature and experiment essence.

This new application refers to the marksmanship game that, in general, is an environment where a thrower is positioned at a given distance from a target and he tries to hit the target by firing a projectile. In this paper, the game is adapted once the throwers are represented by archers that try hitting the targets composed of black or gray apples, as illustrated in **Figure 2** with two scenarios.

In the 1st scenario, there is an archer for each apple, both identified by the same number. Apples are positioned in a parallel line with respect to archers, as shown in **Figure 2(a)**. In an ordered manner, each apple is shot by the respective archer during each iteration of each phase, if the suitable condition is true. In each iteration, it is evaluated the color and status of each apple and the status of each archer.

The conditional evaluation is illustrated in the causal expression in **Figure 3**. The condition is true if the apple color is black, the apple status is ready to be hit, and the respective archer status is ready to shoot the apple. Still,

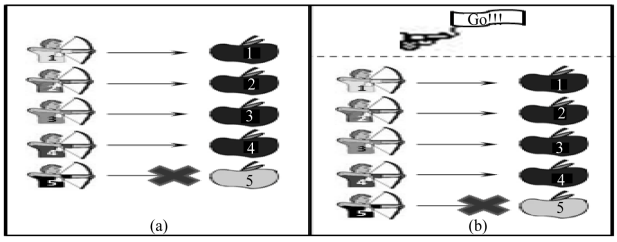

Figure 2. First (a) and second (b) scenario for the marksmanship game.

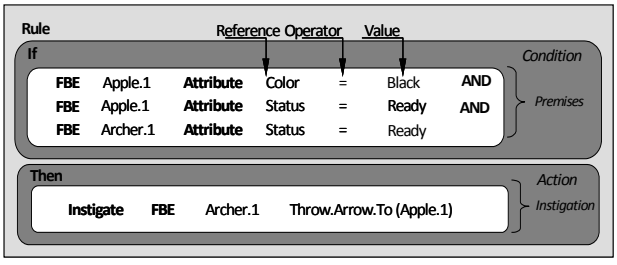

Figure 3. The representation of a rule.

the status of each Apple and the status of each Archer are fixed as ready.

Still in the 1st scenario, before the beginning of each iteration, all apples are gray (*i.e*. disabled). However, in the beginning of each iteration, a given percentage of the apples is replaced by black apples (*i.e*. enabled apples). This percentage is defined and incremented at each phase that the iteration pertains. Still, for an iteration to be completed, each enabled apple must be shot by its respective archer. After completing the iteration, shot apples are replaced by new gray apples in order to be used in the next iteration.

In the 2nd scenario illustrated in **Figure 2(b)**, in turn, interactions between archers and apples present similarities with the 1st scenario. In fact, they differ just in two aspects:

- The addition of a gun to signalize the each iteration start;
- Shot apples are not replaced by new ones in the iterations of a phase because the apples accept more then one shot.

In order to offer a suitable comparison on these scenarios, the experiments vary the amount of satisfied conditional-causal evaluations by phase in each experiment. The percentage of enabled apples (used in the iterations) is varied from none to all in the phases, creating different types of iterations. The aim is to evaluate redundancy effects when the number of causal evaluations in true states is increased.

These scenarios offer a suitable scope to perform comparative tests between Imperative Paradigm and NOP. The first and second scenarios respectively emphasize temporal and structural redundancies on the causal expressions of Imperative languages. Still, they emphasize the main NOP features and advantages in the redundancy removal.

4. 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].

4.1. NOP Structural View

NOP causal expressions are represented by common causal rules, such as that in **Figure 3**, which are naturally understood by programmers of current paradigms. However, each rule is technically enclosed in a special computational-entity called "Rule". An example of Rule Entity content is illustrated in **Figure 4**. This Rule structures and infers the causal knowledge with respect to the case in which an Apple would be crossed by an Arrow projected by an Archer.

Structurally, a Rule has two parts, namely a "Condition" and an "Action", as shown by means of the UML class diagram in **Figure 5**. Both are entities that work together to handle the causal knowledge of the Rule. 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, such as "Apple" and "Archer".

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", such as Color and Status Attributes of the Apple FBE.

Attributes states are evaluated in the Conditions of 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: (a) Is the Color of the Apple Black? (b) Is the Apple Status Ready? (c) Is the Archer Status Ready?

When each Premise of a Rule Condition is in true state,

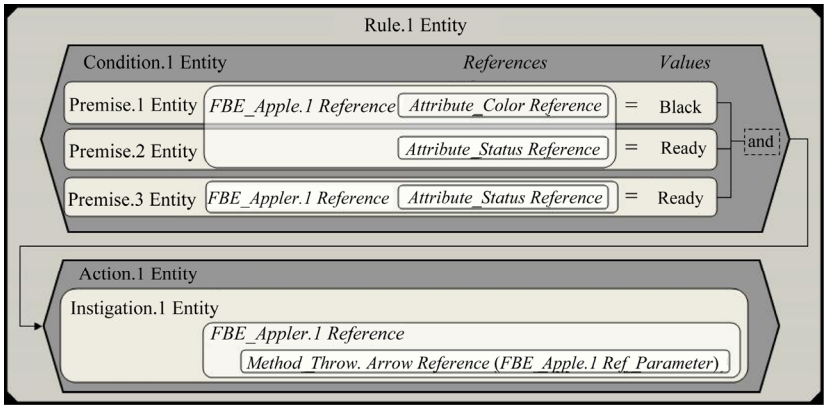

Figure 4. Rule entity.

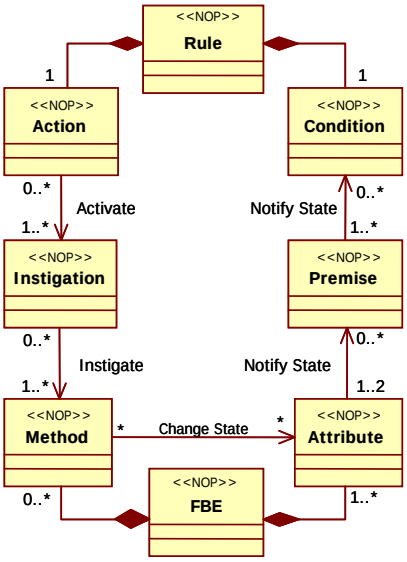

Figure 5. Rule and fact base element class diagram.

which is concluded by means of a given inference process, the Rule becomes true and can activate its Action that is composed of special-entities called “Instigations”. In the considered Rule, the Action contains only one Instigation that makes the Archer to throw an arrow in the Apple.

In fact, 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, feeding the inference process.

4.2. 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 6**. 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.

4.3. NOP Redundancy Avoidance-Performance

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

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 evaluated or reevaluated.

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.

4.4. 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

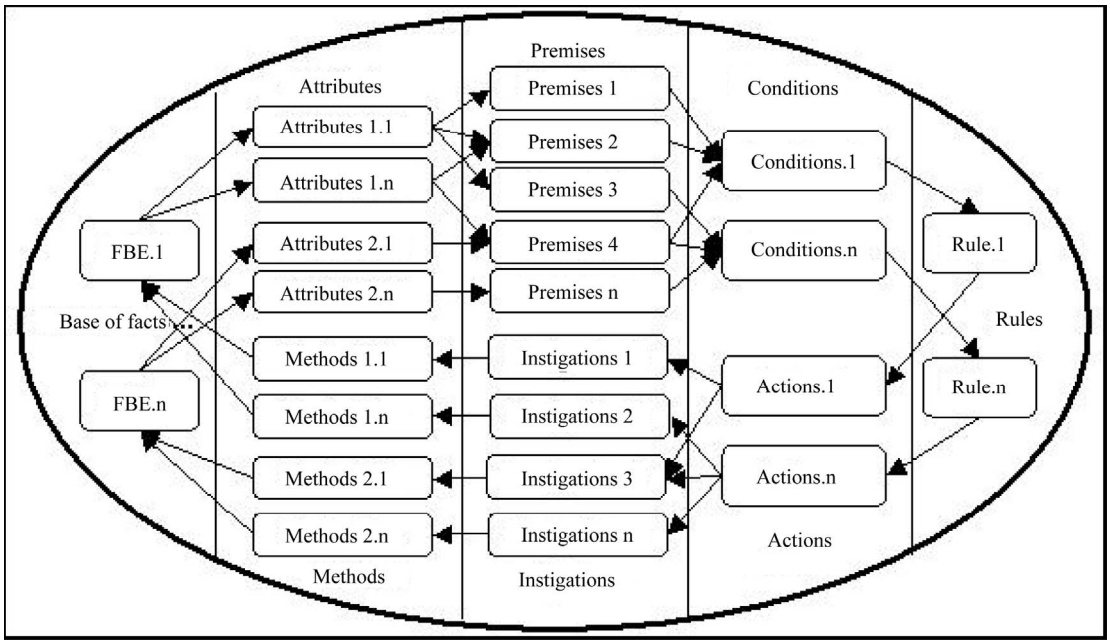

Figure 6. Notification chain of rules and collaborators [3].

computer memory or in the same sub-network.

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.

4.5. 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.

5. A Performance Study

NOP promises to solve some current paradigm deficiencies, highlighting here the imperative one. This promise happens by the proposition of an alternative manner to create and execute software based on an inference mechanism composed of collaborative notifier objects [3,7, 8].

The cooperation between NOP entities (a sort of smart-objects in the current framework) via notifications happens efficiently and in a decoupled way. This allows enhancing application performance in monoprocessed architectures and presumably also in parallel/distributed ones [3,7,9].

This paper emphasizes monoprocessed architectures, presenting NOP as a solution that avoids temporal and structural redundancies, thereby saving resources and speeding up the application performance with respect to imperative programming in terms of causal calculation.

In order to clarify the NOP efficiency, this section presents a performance comparative study between programs from NOP and Imperative Paradigm, which are applied to the two before described marksmanship scenarios. The first scenario highlights temporal redundancies, whereas the second one highlights structural redundancies.

Besides, both scenarios are implemented via C++ language and NOP C++ framework, which are compiled with the compiler DJ's GNU Programming Platform (DJ GPP) to non-preemptive MS-DOS using maximum optimization.

5.1. Scenario Organization

In both scenarios, it is considered that 100 archers interact with the same amount of apples. The archers are rep-

resented by the class Archer and the apples by the class Apple. The Archer and Apple instances are assembled in two vectors, which are respectively named Archer List and Apple List. Respectively to the first and second scenarios, the **Figures 7** and **10** present part of these codes that use vector class of C++ Standard Template Library (STL).

In the IP implementation, two vectors with typical objects are used to define and evaluate 100 imperative causal expressions in the form of *if* statements via a loop. In the NOP implementation, in turn, other two vectors with FBE objects are used only to instantiate 100 NOP causal expression entities, *i.e.* NOP Rules, once Rule evaluations occur via notifications. Of course, both IP and NOP codes have their causal evaluation composed with quite similar efforts in order to carry out a fairer comparison.

In both implementations, there are 10 phases, each one with 100.000 iterations. Each interaction evaluates causal expressions in the concerned paradigm. Still, each phase defines percentage of the causal expressions in true state. First phase defines 10%, second defines 20%, and so forth. Besides, each program was executed 10 times by phase, resulting in an average of used time in milliseconds.

The time is measured only to the iteration execution, ignoring the time related to the preparation to the iterations. In the 1st scenario, this means that the time to replace the shot apples by new ones is ignored. In the 2nd scenario, where the gun should be firstly turned off in order to fire again, the time to reactivate the gun is ignored.

5.2. First Scenario

The causal expressions related to the first scenario are illustrated in the code shown in **Figure 7**, where lines 1 to 10 refer to imperative code and lines 11 to 20 to NOP code.

Each causal expression is composed of three logical expressions or *premises* that respectively refer to the evaluation of: (a) an Apple color (*i.e.* lines 4 and 15), (b) an Apple position status (*i.e.* 5 and 16), and an Archer status (*i.e.* 6 and 17). In these causal expression *premises*, just the first can vary its logical state during the iterations, whereas other two premises always present true logical state.

These causal expressions are used to express the problem of temporal redundancy in Imperative Programming and the solution to this problem in NOP. The temporal redundancy happens when at least one premise is false and it continues false and evaluated in more than one iteration.

In this scenario, considering none causal expression is satisfied in each iteration (*i.e.* 0% of all causal expressions), IP mechanism evaluates and reevaluates indefinitely the causal expressions until a given stop criterion is achieved.

Also, considering only one causal expression satisfied in each iteration (*i.e.* 1% of all causal expressions), IP mechanism evaluates all causal expressions (almost all unnecessarily) in order to execute the commands of the unique satisfied causal expression. This sequential execution process delays the evaluation of pertinent causal expression.

NOP does not use computational resources unnecessarily when there is state invariability, avoiding causal expression reevaluation. Also, it avoids searches notifying punctually each causal expression really affected by state changes and immediately after the change has occurred.

In order to confirm this fact, a practical experiment was carried out to the current scenario. Results are favorable to NOP and are presented in **Figure 8**. According to the graphic, NOP presents better performance when it avoids unnecessary processing and searches for causal expressions.

```
 1 for(int i=0; i < 100; i++)
 2 {
 3   if( (appleList->at(i)->color   == "RED") &&
 4       (appleList->at(i)->status  == true)  &&
 5       (archerList->at(i)->status == true))
 6   {
 7     appleList->at(i)->isCrossed = true;
 8   }
 9 }
10 ...
11 for (int i=0; i < 100; i++)
12 {
13   Rule* rlFireApple = new Rule(Condition::CONJUNCTION);
14   rlFireApple->addPremise(appleList->at(i)->atAppleColor,    "RED",         Premise::EQUAL);
15   rlFireApple->addPremise(appleList->at(i)->atAppleStatus,   Boolean::TRUE, Premise::EQUAL);
16   rlFireApple->addPremise(archerList->at(i)->atArcherStatus, Boolean::TRUE, Premise::EQUAL);
17   rlFireApple->addInstigation(appleList->at(i)->atAppleIsCrossed, true);
18   rlFireApple->end();
19 }
20 ...
```

Figure 7. Causal expressions related to the 1st scenario.

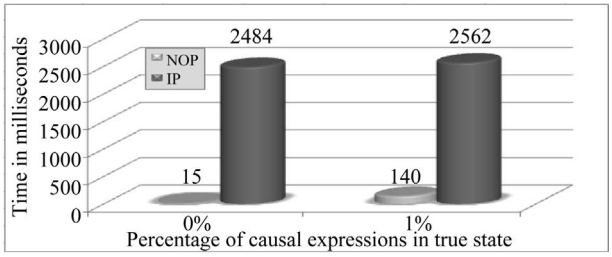

Figure 8. No one and one causal expression in state true.

In agreement with Forgy's research [26], most causal expressions in imperative code are evaluated unnecessarily, once few of them are affected for changes in data during an iteration of the Imperative Paradigm mechanism.

According to him, less than 1% of the application data changes in one iteration. Thus, comparing this affirmation with the actual experiment results to this scenario, NOP can be considered as a solution to improve the efficiency of the most computational systems based on the IP.

However, this affirmation is not applicable to all computational systems. For example, Friedman-Hill [51] affirms that 20% of causal-expression can be affected in each interaction. Still, even if it is not usual, some systems could have, in the worst case, the most or even all causal expressions affected by changes of one or more variable states.

Thus, another experiment was carried out over the present scenario in order to compare both paradigms in relation to different levels of temporal redundancy. In this experiment the causal expressions affected by state changes increase in terms of quantity in each experiment phase in order to reduce the effects of temporal redundancies. These experiments results are expressed in **Figure 9**.

According with the result of this particular scenario, NOP outperforms IP until when 40% of causal expressions are affected by states change. This rate is extremely greater than the rate affirmed by Forgy [26]. However, NOP is less efficient than IP when the changes in the states affect a rate greater than 40%. This happens due to the simplicity that IP evaluates the causal expressions, aided by the reduction of the temporal redundancies in each experimental phase.

This simplicity may be clearly noted by the representations of the causal expressions. In the considered IP code, each causal expression manipulates only two objects (instances of classes Archer and Apple) whereas a NOP Rule manipulates also the collaborator objects from the notification mechanism (Attributes, Premises and Condition).

In the actual NOP implementation, in terms of instructions in Assembly Language, Rules are surely more complex and composed of a greater number of instructions to be processed than causal expressions of Imperative Programming. Furthermore, NOP is currently implemented as an abstraction layer over C++ language, affecting its performance. Thus, the results may be improved by a sequence of optimizations that pass certainly through the construction of a particular compiler [9].

5.3. Second Scenario

The 2nd scenario presents temporal and structural redundancies, but the structural redundancy is highlighted. In this scenario, the gun is a common element to all archers, once these must "listen" the gun signal to hit the apples.

Thus, in each iteration, the gun state must be evaluated by each archer. This fact explains the need of one more premise to the causal expressions presented in the first scenario. The causal expressions with additional premise (line 18) are presented in **Figure 10**.

Considering the causal expressions in the **Figure 10**, Imperative Paradigm mechanism evaluates the premise

True Rules	NOP (ms)	IP (ms)
10%	1517	3170
20%	2847	3949
30%	4595	4858
40%	5866	5877
50%	7608	6890
60%	9435	7858
70%	11043	8795
80%	11874	9713
90%	13825	10657
100%	15775	11604

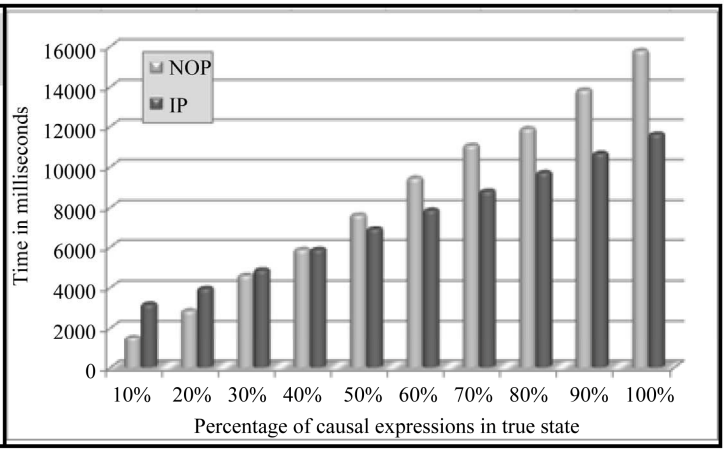

Figure 9. Variable percentage of true causal expressions—1st scenario.


```
 1 for(int i = 0; i < 100; i++)
 2 {
 3   if( (appleList->at(i)->color   == "RED") &&
 4       (appleList->at(i)->status  == true)  &&
 5       (archerList->at(i)->status == true)  &&
 6       (gun->status == true))
 7   {
 8     appleList->at(i)->isCrossed = true;
 9   }
10 }
11 ...
12 for ( int i = 0; i < 100; i++)
13 {
14   Rule* rlFireApple = new Rule(Condition::CONJUNCTION);
15   rlFireApple->addPremise(appleList->at(i)->atAppleColor,    "RED",         Premise::EQUAL);
16   rlFireApple->addPremise(appleList->at(i)->atAppleStatus,   Boolean::TRUE, Premise::EQUAL);
17   rlFireApple->addPremise(archerList->at(i)->atArcherStatus, Boolean::TRUE, Premise::EQUAL);
18   rlFireApple->addPremise(gun->atGunStatus,                  Boolean::TRUE, Premise::EQUAL);
19   rlFireApple->addInstigation(appleList->at(i)->atAppleIsCrossed, true);
20   rlFireApple->end();
21 }
22 ...
```

Figure 10. Causal expressions related to the 2nd scenario.

relative to the Gun state in each causal expression (line 6), whereas NOP mechanism identifies redundancies and shares the same Premise by the respective Conditions (line 18).

Certainly, the imperative code could be more optimized, such as using a causal expression evaluating the gun state before the loop. However, this would not be possible if the causal expressions were spread in many parts of a complex code, as usual happens in software.

In short, the idea is to highlight the structural redundancy that often occurs in Imperative Paradigm code. Thus, in order to effectively evaluate the structural redundancy in this scenario, it was considered that only the gun state is changed in each iteration. This allows varying the logical state just of the premises related to the gun.

Also, it was considered that a percentage of apples are enabled before the beginning of each experiment phase and they remain in these states until the end of the experiment phase to avoid state changes in other premises. The results of each experiment phase are presented in **Figure 11**.

According to the graph over this scenario, NOP presents better performance when it solves both redundancies, presenting better results than the previous scenario. It is due to the capacity of the notification mechanism to memorize logical states already evaluated and sharing of the logical state of the respective Premise to all connected Conditions.

6. Conclusion and Future Works

This section discusses NOP properties and future works.

6.1. 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 [7-9, 29].

In the current paradigms, the evaluation process is based on monolithic inference-process that performs some sort of search over passive fact-bases (e.g. variables and vector sets) and causal-bases (e.g. if-then statement sets), which generates a set of deficiencies. Precise deficiency examples are the misuse of computation capacity and code coupling that respectively generate degradation of the performance and hardness to develop multi-processed software.

In turn, NOP proposes factual and causal smart-entities named as Fact Base Elements (FBEs) and Rules that are related to other collaborative notifier smart-entities. Each FBE is related to Attributes and Methods, whereas each Rule to Premises-Conditions and Actions-Instigations. All these entities collaboratively carry out the inference process by means of notifications, providing solutions to deficiencies of current paradigms. In this context, this paper addressed the performance subject making some comparisons of NOP and Imperative Programming instances.

6.2. NOP Performance

As demonstrated, NOP improves performance 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]. Also, NOP improves the performance by sharing the results of logic evaluation (*i.e.* notification of Premises) between causal evaluations (*i.e.* execution of Conditions), avoiding unnecessary repetitions

True Rules	NOP (ms)	IP (ms)
10%	2790	2985
20%	3002	3233
30%	3382	3531
40%	3516	4094
50%	3828	4626
60%	4122	5187
70%	4371	5687
80%	4747	6250
90%	5307	6734
100%	5564	7311

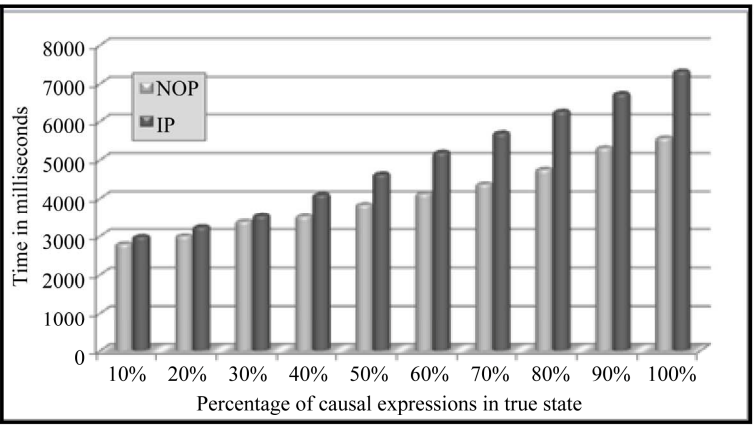

Figure 11. Variable percentage of true causal expressions—2nd scenario.

of code and processing in the execution of the Rules [3].

Thus, temporal and structural redundancies are avoided by NOP, guarantying suitable performance by definition [3]. Actually, even if NOP uses high-level concepts inspired from Rule Based System and Object Oriented System concepts (e.g. rule-objects) and even if its actual implementation is an abstraction layer over C++, NOP implementation does have suitable performance [9].

In this context, under the same conditions, NOP programs presented in this paper outperforms pure Imperative Paradigm programs in relevant percentages of the considered experiments. Besides, other additional experiments with similar results are presented in [9]. Still, some experiments therein compared the NOP implementation with a Declarative Programming best practice and demonstrated its performance superiority [9].

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.

6.3. 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.

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.

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.

6.4. NOP New Paradigm

Even if the causal programming can be easily made using NOP, highlighting the support of a wizard tool, it is necessary to know NOP principles [8]. It is necessary to the developer to understand the structure and execution process of the application under NOP to elaborate better solutions.

This awareness allows understanding, for example, that NOP software has high performance by definition, in term of logical-causal calculation, and that the concerned mechanism is automatically built in background during the causal code composition. This would allow the developer employing this type of code as much as necessary to each application without strong response-time concerns. Still, that awareness allows realizing other possibilities of NOP use, such as understanding how to suitably apply mechanisms to solve conflicts and guaranty determinism.

In this context, a simple mechanism to deal with conflict and determinism is the first Rule approved to be executed. However, there are better mechanisms possible, due to the inference based on notification that can be only well used by developer with understanding of the software paradigm. These issues are somehow described

in embryo in the first author's Ph.D. thesis [12] and last author's M.Sc. Thesis [9] and actually described in two patent requests.

6.5. NOP Decoupling, Distribution & Formalism

The understanding of the NOP nature by the developer normally allows its better use also in the case of distributed software. For example, it is important to know how NOP works in order to find better distribution strategies, such as to allocate together NOP entities having more interactions thereby avoiding unnecessary network communication [3].

Still, the understanding of NOP principles is important to complex applications where the notification flow is intense and need more formalism and traceability, such as real-time discrete-control applications. Indeed, this sort of application may demand support of formal tools to design.

A particular example of formalism is the Petri nets. Actually, Petri nets are compatible with rule-based system in general in terms of expression of causal relations [52]. Moreover, they are particularly compatible with the NOP principles also in term of their essence [3]. In this context, it would be necessary to know NOP and Petri nets principles, understanding that both are naturally compatible [3].

Actually, Petri nets present a manner to model causal relations based on sensitization, which is similar to notification principles of NOP. Even if Petri nets do not precise how really carry out the causal calculation based on sensitization, it presents a model where abstractly this calculation would be notification-driven. At the best of authors' knowledge, there was not a suitable computational solution to really implement and play Petri nets until NOP solution. Before, Petri nets have been misused in computational implementations, which are based on searches and not on notifications or sensitization-like [3,12].

6.6. NOP Expectation

The presented programming solution called NOP is seen as a new paradigm because it provides a new structure, execution approach, and thinking with respect to software development. Besides, NOP is better in many aspects than the current paradigms and can be used together with them.

Thus, the applicability of NOP concepts presents expectation. It is believed that the maturity of the solution would allow its diffusion and adoption as an alternative to increase the performance and make better the conception of software in non-distributed and distributed environments.

7. Acknowledgements

R. F. Banaszewski's M.Sc. thesis [9] was supported by CAPES Foundation (Brazil).

REFERENCES

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

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

[3] J. M. Simão and P. C. Stadzisz, "Inference Based on Notifications: A Holonic Metamodel Applied to Control Issues," *IEEE Transactions on Systems, Man and Cybernetics, Part A*, Vol. 39, No. 1, 2009, pp. 238-250. 10.1109/TSMCA.2008.2006371

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

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

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

[7] J. M. Simão, 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.

[8] 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," 4*th Congress of Logic Applied Technology*, Santos, 21-23 November 2007, pp. 216-222.

[9] R. F. Banaszewski, "Notification Oriented Paradigm: Advances and Comparisons," M.Sc. Thesis, Federal University of Technology of Paraná, Curitiba, 2009.

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

[11] D. Harel, H. Lacover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauting and M. Trakhtenbrot, "Statemate: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transaction on Software Engineering*, Vol. 16, No. 4, 1990, pp. 403-414. doi:10.1109/32.54292

[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, Federal University of Technology of Paraná, Curitiba, 2005.

[13] B. De 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*, Vol. 3144, 2004, pp. 132-145. doi:10.1007/978-3-540-27860-3_14

[14] 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," *John von Neumann Institute for Computing*, Vol. 38, 2007, pp. 347-354.

[15] W. M. Johnston, J. R. P. Hanna and R. J. Millar, "Advance in Dataflow Programming Languages," *ACM Computing Surveys*, Vol. 36, No. 1, 2004, pp. 1-34. doi:10.1145/1013208.1013209

[16] G. Coulouris, J. Dollimore and T. Kindberg, "Distributed Systems—Concepts and Designs," Addison-Wesley, Boston, 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*), Las Vegas, 13-15 August 2007, pp. 14-15. doi:10.1109/IRI.2007.4296581

[18] 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

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

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

[21] S. H. Kaisler, "Software Paradigm," John Wiley & Sons, Hoboken, 2005.

[22] M. Gabbrielli and S. Martini, "Programming Languages: Principles and Paradigms," Springer-Verlag, London, 2010.

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

[24] 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

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

[26] 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

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

[28] M. L. Scott, "Programming Language Pragmatics," 2nd Edition, Morgan Kaufmann Publishers Inc., Waltham, 2000.

[29] J. M. Simão, C. A. Tacla and P. C. Stadzisz, "Holonic Control Metamodel," *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, Vol. 39, No. 5, 2009, pp. 1126-1139. doi:10.1109/TSMCA.2009.2022060

[30] 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.

[31] S. M. Ahmed, "CORBA Programming Unleashed," Sams Publishing, Indianapolis, 1998.

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

[33] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning," *Lecture Notes in Computer Science*, Vol. 2374, 2002, pp. 178-204.

[34] S. Loke, "Context-Aware Pervasive Systems: Architectures for a New Breed of Applications," Auerbach Publications, Boca Raton, 2006. doi:10.1201/9781420013498

[35] 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

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

[37] H. Tianfield, "A New Framework of Holonic Self-Organization for Multi-Agent Systems," *IEEE International Conference on System, Man and Cybernetics*, Montreal, 7-10 October 2007, pp. 753-758. doi:10.1109/ICSMC.2007.4414048

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

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

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

[41] S. Russel and P. Norvig, "Artificial Intelligence: A Modern Approach: Englewood Cliffs," Prentice-Hall, Upper Saddle River, 2003.

[42] D. P. Miranker, "TREAT: A Better Match Algorithm for AI Production System," 6*th National Conference on Artificial Intelligence*, Seattle, 13-17 July 1987, pp. 42-47.

[43] 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. 1, No. 1, 1991, pp. 3-10. doi:10.1109/69.75882

[44] 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*, Boston, 29 July-3 August 1992, pp. 685-692.

[45] D. Watt, "Programming Language Design Concepts," John Wiley & Sons, Hoboken, 2004.

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

[47] 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.

[48] C. E. B. Paes and C. M. Hirata, "RUP Extension for the Software Performance," 32*nd Annual IEEE International Computer Software and Applications*, 28 July-1 August 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] I. Sommerville, "Software Engineering," 8th Edition, Addison-Wesley, Boston, 2004.

[51] E. Friedman-Hill, "Jess in Action: Rule-Based System in Java," Manning Publications Company, Greenwich, 2003.

[52] V. R. L. Shen and T. T. Y. Juang, "Verification of Knowledge-Based Systems Using Predicate/Transition Nets," *IEEE Transactions on Systems, Man, and Cybernetics, Part A*: *Systems & Humans*, Vol. 38, No. 1, 2008, pp. 78-87.

