Journal of Software Engineering and Applications, 2012, 5, 722-736
http://dx.doi.org/10.4236/jsea.2012.59085 Published Online September 2012 (http://www.SciRP.org/journal/jsea)

A Game Comparative Study: Object-Oriented Paradigm and Notification-Oriented Paradigm

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

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

Received July 13th, 2012; revised August 25th, 2012; accepted September 8th, 2012

ABSTRACT

This paper presents a new programming paradigm named Notification-Oriented Paradigm (NOP) and analyses the performance aspects of NOP programs by means of an experiment. NOP provides a new manner to conceive, structure, and execute software, which would allow better performance, causal-knowledge organization, and decoupling than standard solutions based upon usual paradigms. These paradigms are essentially Imperative Paradigm (IP) and Declarative Paradigm (DP). In short, DP solutions are considered easier to use than IP solutions due to the concept of high-level programming. However, they are considered slower in execution and less flexible in development. Anyway, both paradigms present similar drawbacks such as redundant causal-evaluation and strongly coupled entities, which decrease the software performance and the processing distribution feasibility. These problems exist due to an orientation to a monolithic inference mechanism based upon sequential evaluation by searching on passive computational entities. NOP proposes another way to structure software and make its inferences, which is based upon small, collaborative, and decoupled computational entities whose interaction happens through precise notifications. In this context, this paper presents a quantitative comparison between two equivalent implementations of a computer game simulator (Pacman simulator), one developed according to the principles of Object-Oriented Paradigm (OOP/IP) in C++ and other developed according to the principles of NOP. The results obtained from the experiments demonstrate, however, a quite lower performance of NOP implementation. This happened because NOP applications are still developed using a framework based on C++. Besides, the paper shows that optimizations in the NOP framework improve NOP program performance, thereby evidencing the necessity of developing a NOP language/compiler.

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

1. Introduction

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

1.1. Review Stage

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

This behavior has been tolerated in standard software development where generally 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 [1,4]. Such systems normally employ less-powerful processors due to factors such as constraints on power use and system price to a given market [1,5].

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

Thus, suitable engineering tools for software development, namely programming languages and environments, should facilitate development of optimized and correct code [1,9-12]. Otherwise, the engineering costs to produce optimized-code could exceed those of upgrading the processing capacity [1,4,9-11].

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

In this context, a problem arises from the fact that usual programming languages (e.g. Pascal, C/C++, and Java) present no real easiness to develop optimized and really distributable code, particularly in terms of fine-grained decoupling of code [1,3,4,17,18]. This happens due to the structure and execution nature imposed by their paradigm [1,7,9,10].

1.2. Imperative and Declarative Programming

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

Essentially, Imperative Paradigm imposes loop-oriented searches over passive elements related to data (e.g. variables, vectors, and trees) and causal expressions (*i.e.* if-then statements or similar ones) that cause execution redundancies. This leads to the creation of programs as monolithic entities comprising prolix and coupled code, thereby generating non-optimized and interdependent code execution [1,8,9,21,22].

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

Declarative Paradigm solutions use computationally expensive high-level data structures causing considerable processing overheads. Thus, even with redundant code, Imperative Paradigm solutions are normally better in performance than Declarative Paradigm solutions [1,10, 27]. Furthermore, similarly to the Imperative Paradigm programming, the Declarative one also generates code coupling due to the similar search-based inference process [1,4,8,21]. Still, other approaches such as event-driven and functional programming do not solve these problems even if they may reduce some problems, like redundancies [1,22,27] .

1.3. Development Issues & Solution Perspective

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

The NOP basis was initially proposed by J. M. Simão as a manufacturing discrete-control solution [28,29]. This solution was evolved as general discrete-control solution and then as a new inference-engine solution [4], attaining finally the form of a new programming paradigm [1, 8-10]. Since then, efforts have been produced to the establishment of this paradigm [1,30-36].

The essence of NOP is its inference process based on small, smart, and decoupled collaborative entities that interact by means of precise notifications [1,4]. This solves redundancies and centralization problems of the current causal-logical processing, thereby solving processing misuse and coupling issues of usual paradigms [1,4,8,10].

1.4. Paper Context and Objective

This paper discusses NOP as a solution to certain deficiencies present in usual paradigms. Particularly, the paper presents a performance study, using a single-processor architecture computer.

The study is related to a game simulator implementation based on principles of NOP compared against an equivalent one based on principles of 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 to evaluate the current NOP materialization in terms of performance.

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

Thus, another objective is to demonstrate how relevant and appropriated are some refactoring and optimizations in the NOP Framework, thereby realizing whether it is or not necessary to build a NOP compiler.

2. Background

This section explores programming paradigm drawbacks.

2.1. Imperative Programming (IP) Issues

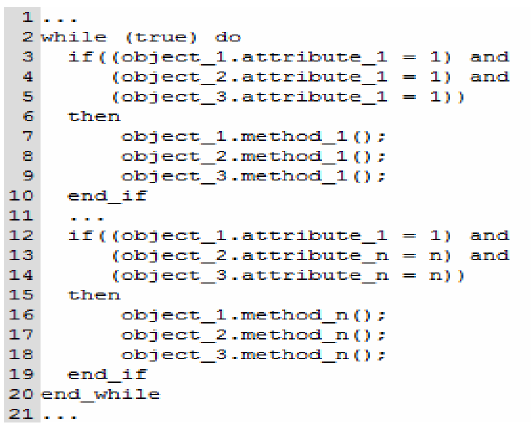
The main drawbacks of Imperative Programming are concerned to the related code redundancy and coupling. The first mainly affects processing time and the second processing distribution, as detailed in the next subsections [1,4].

2.1.1. Imperative Programming (IP) Redundancy

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

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

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



```
...
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 [1].

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 consecutively an increasing of development complexity in order to improve the performance [1,8,10].

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

2.1.2. Imperative Programming (IP) 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 [1,4].

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

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

2.1.3. IP Distribution Hardness

When distribution is intended (e.g. process, processor, and cluster distribution), code analysis could identify less dependent code arrangements to facilitate their splitting. However, this is normally a complex activity due to the code coupling and complexity caused by the imperative programming [1,18,35].

In this sense, well-designed software composed of modules as decoupled as possible, using advanced and quite complicated software engineering concepts such as *aspects* [13] and *axiomatic design* [37], can help distribution. Still, middleware such as CORBA and RMI would be helpful in terms of infrastructure to some types of module distribution, if there is enough module decoupling [1,13,38,39].

In spite of those advances, distribution of single code elements or even code modules is still a complex activity demanding research efforts [13,14,17,35,40]. It would be necessary additional efforts to achieve easiness in distribution (e.g. automatic, fast, and real-time distribution), as well as correctness in distribution (e.g. fine-grained, balanced, and minimal inter-dependent distribution) [1,4].

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

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

2.1.4. IP 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 [1,47]. 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 signifycant easiness in this activity and true fine-grained code decoupling [1].

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 [1,19,48].

2.2. Declarative Programming Issues

A well-known instance of Declarative Programming and its nature is Rule Based System (RBS) [1,4,47]. A RBS provides a high-level language in the form of causal-rules, that prevents developers from algorithm particularities [1,47]. RBS is composed of three general modular entities (Fact Base, Rule Base, and Inference Engine) with distinguished responsibilities, as usual in declarative language (e.g. LISP, PROLOG, and CLIPS) [1,48].

In Declarative Programming, the variable states are dealt in a Fact Base and the causal knowledge in a Causal Base (Rule Base in RBS), which are automatically matched by means of an Inference Engine (IE) [1,24,47]. Moreover, some IE algorithms (e.g. RETE [23-25], TREAT [49,50], LEAPS [51], and HAL [26] algorithms) avoid most of temporal and structural redundancies [1,10]. However, the data structures used to solve redundancies in those IEs implies in too much consuming of processing capacity [1,25].

Actually, the use of Declarative Programming only compensates when the software under development presents many redundancies and few data variation. Also, in general, an IE related to a given declarative language limits the inventiveness, makes difficult some algorithm optimizations, and obscures hardware access, which can be inappropriate in certain contexts [1,10,22,27,52].

A solution to these problems can be the symbiotic use of Declarative and Imperative Programming [19,52]. Indeed, such approach has been presented, like CLIPS++, ILOG Rules, and R++. However, they would not be popular due to factors like syntax and paradigms mixing or technical cultural reasons [10]. Anyway, even Declarative Programming being relevant, it does not solve some issues [1].

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

2.3. Other Approach Drawbacks

Enhancements in the context of Imperative and Declarative Paradigm have been provided to reduce effects of recurrent loops or searches, such as event-driven programming and functional programming [10,48,53]. Event programming and functional programming have been used to different software like discrete control, graphical interfaces, and multi-agent systems [1,10,48,53].

Essentially, each event (button pressing, hardware interruption or 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 would mean procedure, method or similar unity. Besides, functional and event programming used together would be usual [1].

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

An alternative programming approach is the Data Flow Programming [14] that supposedly should allow program execution oriented by data instead of an execution line based on search over data. Thus, this would allow decoupling and distribution [14]. The distribution in Data Flow Programming is achieved in arithmetical processing, however it is not really achieved in logical-causal calculation [14,17]. This calculation is carried out by means of current advanced inference engines, namely Rete [1,17,54].

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

2.4. Enhancement in Programming

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

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

This is a problem mainly when the increasing market demand by software is considered, where development easiness, code optimization, and processing distribution are current requirements [1,55-57]. This software development "crisis" impels new researches and solutions to make simpler the task of building better software [1].

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

3. Comparative Study

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 [28]. Those concepts revealed to be suitable to control applications [28,29].

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 Meta-model 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 application [1,4,8].

Anyway, each control application over ANALYTICE II is actually complex to be used in a comparison study between NOP and Object-Oriented Paradigm (OOP). Indeed, the understanding of this complex application could undermine NOP and experiment understanding as well.

In this context, a simpler application called Marksmanship Game was proposed in a previous work in order to evaluate the paradigm. This toy application has allowed implementation experiments showing good results in favor of NOP [10].

However, other tests on more realistic applications are needed to effectively demonstrate the efficiency and effectiveness of NOP in relation to usual programming paradigms. Thus, another application is here proposed, being called Pacman Game Simulator. This application was firstly described (in Portuguese) in [33] and was there first used to some previous experiments.

3.1. Pacman Game Simulator

Concisely, this new application refers to an implementation of a simulator with features of the classic game knows as Pacman[1]. Resembling the game of its inspiration [58], the environment has corridors that form a maze and limit the movement actions of the characters in the scenario, namely the Pacman and his enemies, the ghosts. Moreover, all the characters present autonomous behavior (*i.e.* they are not controlled by a user). **Figure 2** illustrates the environment produced by the simulator [33].

At the beginning of the simulation, the Pacman is placed at (9, 15) position in the grid, whose origin is located at the upper-left corner at (0, 0) position. In turn, the four ghosts are placed in a prison at the center of the grid and may leave after a given period of time [33].

During the simulation, the Pacman is free to absorb dots that are in the corridors in order to accumulate points. More precisely, there are 146 normal dots and four large flashing dots called energizers. Each normal dot that is absorbed is worth 10 points, whilst the energizers are worth 50 points each. This yields a total of 1660 points for clearing the maze of dots. When the Pacman is accumulating points, it should avoid contact with the ghosts whom try to collide against him [33].

Still, when the Pacman absorbs an energizer, for a short period of time, the "tables are turned" and the ghosts assume the role of prey whilst the Pacman assumes the role of predator. During this period, the Pacman can accumulate points colliding against each of the four ghosts. Colliding with a frightened ghost yields

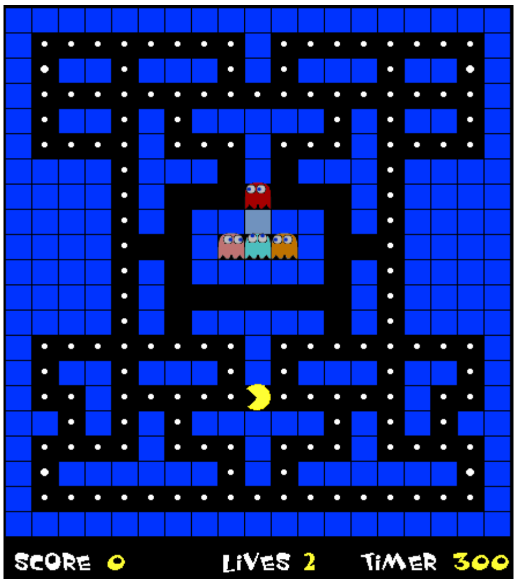


Figure 2. Environment produced by the simulator [33].

[1]The copyrights belong to the individual that produced it or the company that published it. The use in this paper refers only to academic study.

200 points to the Pacman. Each additional collision against a ghost from the same energizer will score the double of the previous collision—400, 800, and 1600 points, respectively [33].

The maximum possible score achieved by the Pacman is 13,660. The Pacman has a certain period of time to accumulate as many points as possible, featuring three lives to do so. Each collision between the Pacman and a ghost out of the energizer's period results in loss of life and a repositioning of the characters in their initial position s [33].

Specially, in this simulator, the characters have a visual field that represents the depth with which they see all the way through the corridors of the maze. **Figure 3** illustrates an example of the visual field with depth 5. The scope of the characters' vision is represented by circles, which is limited when faced with the end of the corridor [33].

3.2. Features of Implementation

The simulator has some special features that benefit the implementation of causal rules for the movement of the characters in the corridors of the maze. Among these peculiarities, the maze has been divided into categories of corners. Each corner represents the meeting or intersection of two or more corridors that compose the maze [33].

The maze consists of 9 different formats of corners in order to minimize the amount of causal rules, since the treatment of characters' actions is based on this classification and not in all parts of the maze. As shown in **Figure 4**, each particular shape of the corner is represented by a value, present on a scale of 1 to 9 [33].

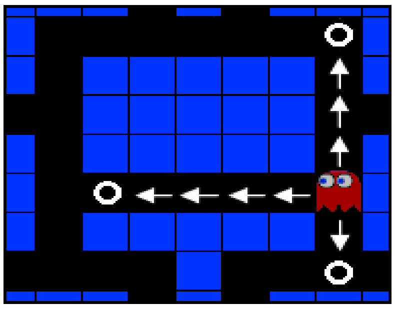


Figure 3. Visual Field with depth of 5 [33].

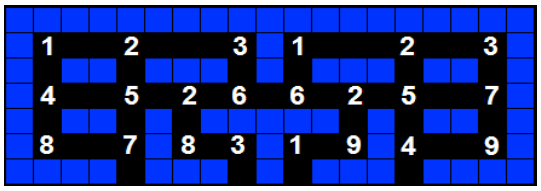


Figure 4. Maze divided into 9 different shapes of corners [33].

As shown in **Figure 3**, characters have a certain vision capability that allows them to perceive elements that compose the maze. For example, the Pacman sees dots, walls, ghosts, and empty corridors. As well as, the ghosts see walls, corridors, the Pacman, and other ghosts. Thus, based on his view, the characters must make consistent decisions to move through the corridors of the maze in search of achieving their goals [33].

The decisions of the characters with respect to their movement's actions are dependent on their perception of elements in their visual fields. For each character, events are fired as they detect elements in their visual fields. Usually more than one element is detected every time a character is in a corner. In order to avoid conflicts, only one event can be active at any instant, being necessary to define a scale of priorities for such events [33].

The events instigate the movement actions of the Pacman have the following order of priority [33]:

- Ghost detected in normal state.
- Ghost in the scared state detected.
- Dot detected.
- Empty corridor.

On the other hand, events that instigate movement actions of the ghosts have the following priority order [33]:

- Pacman detected whilst the ghost in scared state.
- Pacman detected whilst the ghost in normal state.
- Another ghost detected.
- Empty corridor.

The characters only change their direction when they are on a corner, with the possibility of return to the path by which they came or choose another corridor to follow. **Figure 5** illustrates an example of a causal rule that moves the character Pacman [33].

The causal rules that move the characters in each particular situation are predefined by the simulator user (*i.e.* developers) or with the help of learning algorithms (e.g. genetic algorithms), which would generate causal rules for each of the characters that compose the environment [33].

In this paper, a set of causal rules was elaborated by developers. These causal rules were respected in both implementations (*i.e.* implementations in NOP and OOP) followed the given causal rules. Moreover, characters have the same initialization in all implementations.

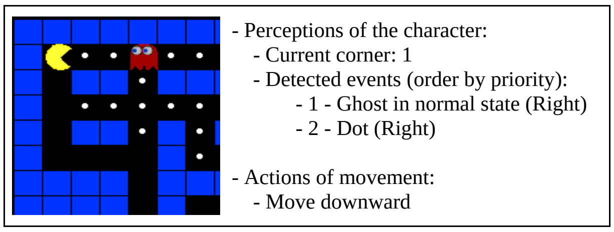

- Perceptions of the character:
- Current corner: 1
- Detected events (order by priority):
- 1 - Ghost in normal state (Right)
- 2 - Dot (Right)
- Actions of movement:
- Move downward

Figure 5. Example of a causal rule that moves the Pacman [33].

These aspects together result, in terms of characters actions, in a specific and equal execution in every run of the simulator, both in NOP and OOP implementations.

4. Notification Oriented Paradigm (NOP)

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

4.1. NOP Structural View

NOP causal expressions are represented by common causal rules, which are naturally understood by programmers of usual paradigms. However, each causal rule is technically enclosed in a special computational-entity called "Rule" [1]. An example of Rule Entity content is illustrated in **Figure 6**. This Rule structures represent the causal knowledge related to the case in which the Pacman should avoid collision with ghost by moving downwards.

Structurally, a Rule has two parts, namely a "Condition" and an "Action", as shown by means of the UML class diagram in **Figure 7**. 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, such as "Creature" and "Dot".

NOP factual elements are represented via a special type of entity called "Fact_Base_Element" (FBE). A FBE includes a set of attributes [1]. Each attribute is represented by another special type of entity called "Attribute", such as "CurrentCorner" and "ElementDir" Attributes of the Creature FBE.

Attributes states are evaluated in the Conditions of Rules by associated entities called "Premises" [1]. In the example, the Condition of the Rule is associated to three Premises, which verify the state of FBE Attributes as follow: 1) Is the Pacman in the current corner 1? 2) Is the detected element a Ghost? 3) Is the element direction in the Right?

When each Premise of a Rule Condition is in true state, 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" [1]. In the considered Rule, the Action contains only one Instigation that makes the Pacman moves down.

In fact, Instigations are linked to and instigate the

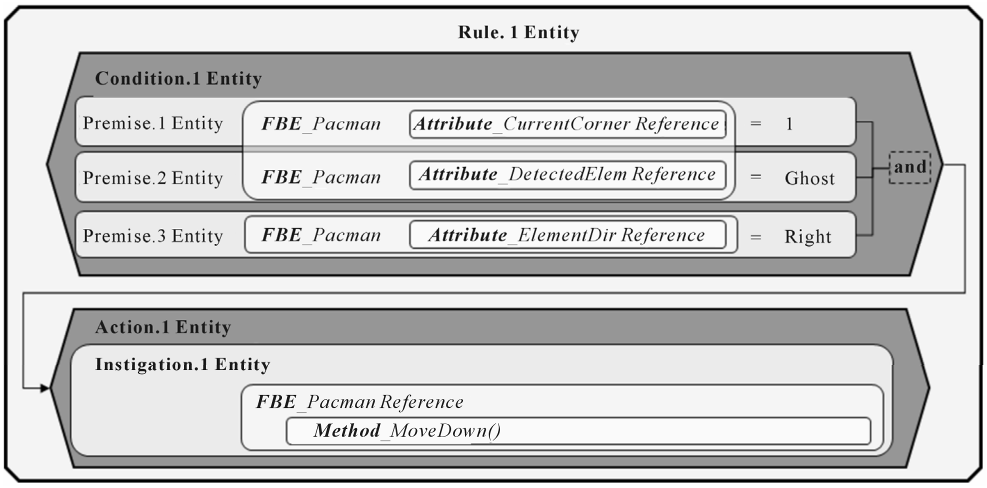


Figure 6. Rule entity.

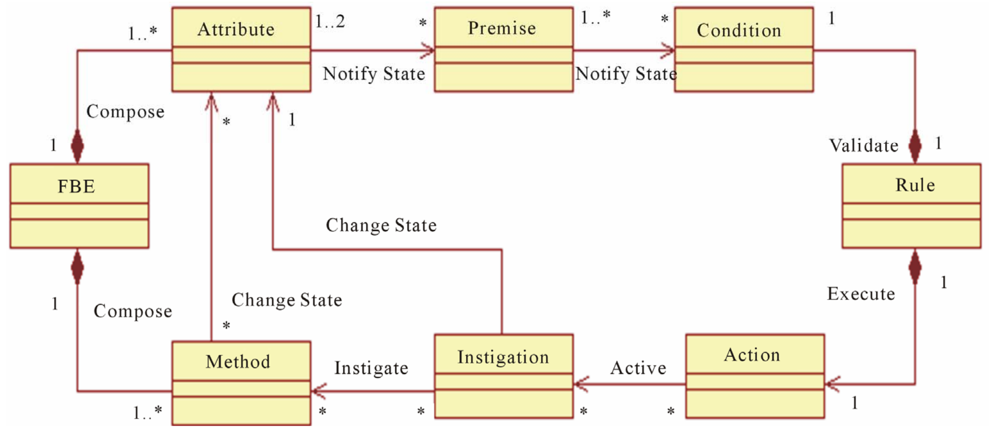


Figure 7. Rule and Fact_Base_Element class diagram [33].

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 [1].

4.2. NOP Inference Process

The NOP inference process is innovative once the Rules have their inference carried out by active collaboration of its notifier entities. 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 punctual notifications between the collaborators [1,4].

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

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 [1,4].

Thus, each notified Condition calculates their Boolean value by the conjunction of Premises values. When all Premises of a Condition are satisfied, a Condition is also satisfied and notifies the respective Rule to execute [1,4].

The collaboration between NOP entities by notifications can be observed at the schema illustrated in **Figure 8**. In this schema, the flow of notifications is represented by arrows linked to rectangles that symbolize NOP entities

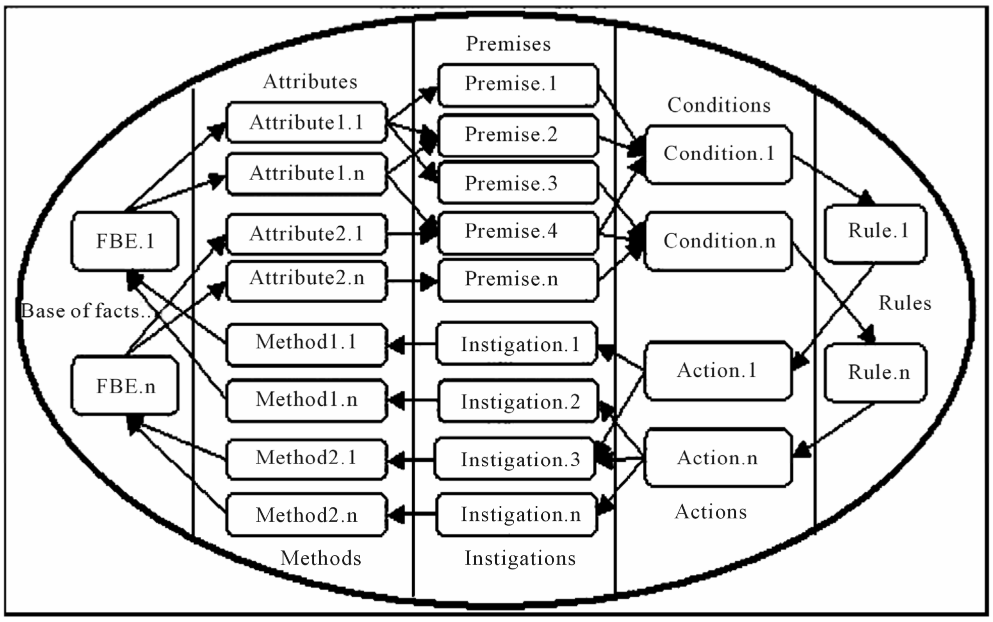


Figure 8. Notification chain of Rules and collaborators [1,4,8].

[1,4].

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

4.3. NOP 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 [1,4].

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

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

4.4. NOP—Decoupling and Distribution

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

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

4.5. NOP Originality

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

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

Indeed, this inference innovation changes all the software essence with respect to logical-causal reasoning (one of its essential parts) and then makes the solution a new programming paradigm. Moreover, as NOP changes the form that software is structured and executed, it also determines a change in the form that software is conceived [1,10,30].

4.6. NOP Implementation

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

Anyway, the development of NOP applications have been made just by instantiating the framework [10, 30-34,36]. Moreover, to make easier this process, a prototypal wizard tool was implemented to automate this process. This tool generates NOP smart-entities from causal rules elaborated in a graphical interface.

In this case, developers "only" need to implement FBEs with Attributes and Methods (with already some help from the NOP wizard tool), once other NOP special-entities will be actually composed and linked by the tool. Thus, the programmer can make use of the time to construct the causal base (*i.e.* composition of NOP Rules) without concerns to instantiations of the NOP Rules.

Nevertheless, current applications developed in NOP actually run over some extra layers due to the NOP Framework materialization over the C++ programming language. Thus, this framework has been constantly optimized to improve the performance of NOP applications.

Among all optimizations performed on the Framework, the part of the inference process was most impacted. In its original artifact designed by Banaszewski [10], the *Original NOP Framework* was composed of data structures that hold and iterate over elements. These structures are vector and list of the C++ Standard Template Library (STL).

Such structures are considered high level and affect the performance of the inference process. Thus, particular (*i.e.* "in-house") data structures were implemented in order to improve the execution of the NOP inference process, as discussed in [34], thereby generating the so-called *Optimized NOP Framework*.

5. A Performance Study

NOP proposes to solve some deficiencies of the usual programming paradigms, highlighting here some deficiencies of the Object Oriented Paradigm (OOP). This promise happens by the proposition of an alternative manner to create and execute software based on an inference solution composed of collaborative notified objects [1,4,8,9].

The cooperation between NOP entities (a sort of smart-objects in the current framework materialization) via notifications happens efficiently and in a decoupled way. This would allow enhancing application performance (in terms of causal calculation) in single-processor architectures and presumably also in parallel/distributed ones [1,4,8].

This paper emphasizes single-processor architectures, presenting NOP as a solution that avoids temporal and structural redundancies, thereby supposedly saving resources and speeding up the application performance with respect to imperative programming in terms of causal calculation [1,4].

In order to clarify the NOP efficiency, this section presents a performance comparative study via implementations of the Pacman game simulator under the principles of the OOP and under the two different implementations of the NOP Framework, namely the original NOP framework [10] and the optimized NOP framework [34].

5.1. Paper New Experiment

This paper presents new experiments with respect to previous publications. Particularly, there are differences concerning the experiment developed in [33] (and presented in Portuguese) and this here developed (and presented in English).

A first difference is that in [33] the optimized NOP framework was in a prototypal release, whereas in this current paper the framework was in a more advanced release. In the prototypal release, the notification process in each notifier entity (e.g. each Attribute, each Premise etc) was based on "in house" linked list. Nowadays, in the current release, the notification process in each notifier entity is based on quite optimized dynamically allocated vector, which presents better results.

Also, the experiment concerning the prototypal release of the optimized NOP framework and the original NOP framework did not use the Premise sharing (in both frameworks). Thus, the NOP code therein presented structural redundancies. In turn, the current experiment shares Premises (in both frameworks), thereby preventing structural redundancies.

Still, there are some minor differences in the Rules of the NOP code elaborated in the previous experiment with respect to the current one. Moreover, the previous paper presents fewer details about the experiment implementation than this current paper.

Moreover, the experiment concerning the prototypal release of the optimized NOP framework was done under Linux *Ubuntu* operating system without real concerns with respect to preemption. In turn, the experiment concerning the current release was done under Linux *Debian* operating (a "lean" Linux version) with minimum of preemption.

Besides, implementations of the Pacman simulator in both paper were done by using C++ language and NOP C++ Frameworks, which were compiled with the GCC compiler (in standard compilation mode). Still, in both papers, the used computer was a notebook containing AMD Turion X2 processor with 3 Gigabytes of RAM memory and 2.0 GHz of clock.

5.2. Structural Modeling—Class Diagram

Due to the dimensions of the original diagram, the class diagram shown in **Figure 9** abstracts the complexity of the simulator, expressing its essence by means of the core classes. The class Game contains the functional structure of the simulator, consisting of classes Maze, Score, and Timer. The class Maze represents the structure of the labyrinth formed by walls (Wall), corners (Corner), dots (Dot) and characters (Character) [33].

The core of the simulator, in turn, consists of causal rules that move the characters in the corridors of the maze, concentrated in the Character subclasses. In short, each Ghost has inside its causal rues, as well as has the Pacman.

Still, in general, the simulator is developed under the principles of OOP. For the NOP version of the Pacman simulator, only the part responsible for the decision-making of the characters is implemented following the principles of the NOP programming [33].

5.3. Computational Experiments

Computational experiments were undertaken in order to verify the performance difference between the different implementations of the simulator. For these, tests were performed with the implementation developed in OOP and with the implementation developed in NOP under the two different stable versions of the NOP Framework namely the original NOP framework and the optimized NOP framework (this in a current stable release).

The implementation structure adopted for the OOP version of the simulator is composed of decision-making structures (*i.e.* ifs) that represent the characters' perceptions on the environment (*i.e.* movement policy). In order to explore some deficiencies of the usual programming languages, such as structural redundancy, causal rules of movement were developed in OOP with some usual repetitive and unnecessary evaluations. The **Pseudocode 1** represents this sort of OOP code for Pacman as example.

In turn, in NOP, the implementation adopted for the simulator consists of a set of Rules and its sub-entities. In each Rule, the Condition represents the set of character's perceptions at a given time, with the premises representing the current corner, detected element, and the location of such element. In turn, the respective Action is composed of an Instigation that activated determined Method, resulting in the suitable character's movement.

Pseudocode 2 expresses in the form of causal rules the essence of the program composed in NOP with *Rules* and their collaborative entities. As previously mentioned, the

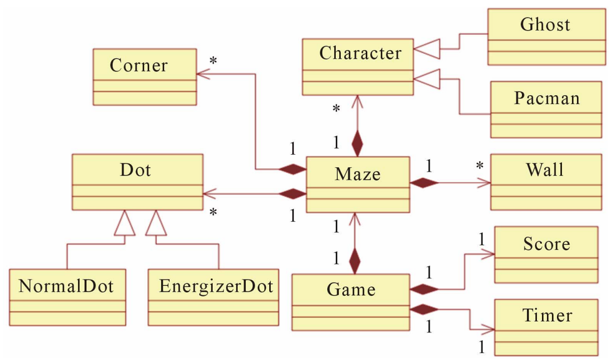


Figure 9. Class diagram of the simulator [33].

```
If ( (Corner = 1) and (Visual Field = Ghost in normal state) and
(Detected Element Direction = Right) )
then moveDownwards()
endif

if ( (Corner = 1) and (Visual Field = Ghost in scared state) and
(Detected Element Direction = Right) )
then moveRight()
endif
. . .
if ((Corner = 9) and (Visual Field = No element detected)) then
moveLeft()
Endif
```

Pseudocode 1. Pacman's rules of movement developed in OOP.

```
Rule 01: Premise – Corner = 1
         Premise – Visual Field = Ghost in normal state
         Premise – Detected element direction = Right
         Instigation–Instigates Method moveDownwards()
Rule 02: Premise – Corner = 1
         Premise – Visual Field = Ghost in scared state
         Premise – Detected element direction = Right
         Instigation – Instigates Method moveRight()
. . .
Rule 80: Premise – Corner = 9
         Premise – Visual Field = No element detected
         Instigation – Instigates Method moveLeft()
```

Pseudocode 2. Pacman's rules of movement developed in NOP.

instance of a specific Character subclass (*i.e.* Pacman or Ghost) is responsible for its own movement in the corridors of the maze. Thus, all conditional structures in OOP and NOP Rules are in each Pacman or Ghost instance.

The two versions of the simulator were implemented in an equivalent manner based on the principles of OOP and NOP. The tests were executed 1, 10, 100 and 1000 iterations over the simulator. Each iteration consists of a complete run of a Pacman game simulation, according to the causal rules defined in the beginning of the execution.

In order to provide reliable performance tests, the predefined causal rules are the same for all the experiments. In order to ensure the accuracy of the data, the same experiments were performed with 100 repetitions each. Thus, the data shown in **Table 1** represents the average time of the experiments in microseconds. The results are discussed in the subsequent subsection.

5.4. Computational Experiments and Results

Table 1 presents results of the implementations based on the causal rules expressed in the **Pseudocodes 1** and **2**. Besides, the **Table 2** presents results of quite similar implementations comparing the original NOP framework and the optimized NOP framework in a prototypal release. This implementations and results were previously published in [33], as aforementioned.

It is possible to observe that the runtime difference between the implementations shows that OOP presents better results compared to NOP, in the two framework versions, namely the original NOP framework and the optimized NOP framework both in prototypal and stable release.

Table 1. Current experiment results—OOP × NOP in *Debian* OS.

Iterations	OOP	NOP (Original)	NOP (Optimized—Stable Release)
1	2312	19,395	3935
10	23,123	195,123	38,978
100	234,527	1,945,285	389,913
1000	2,384,329	19,756,192	3,902,992

Table 2. Previous results—OOP × NOP in *Ubuntu* OS [33].

Iterations	OOP	NOP (Original)	NOP (Optimized—Prototypal Release)
1	269	4233	1418
10	2703	42,598	14,212
100	26914	4.24429	1.42199
1000	2.69858	42.41222	14.31158

The current stable optimized NOP framework release seems better than the prototypal release. However, even been the experiments somehow similar in functional terms, the experiment in [33] has differences from the current one, inclusively in terms of Operating System (OS) as detailed in subsection 5.1. Thus, the data in **Table 2** are not accurate to clearly establish the real difference between prototypal and stable releases of the optimized NOP framework.

Nevertheless, there is a considerable difference between the original NOP framework and the optimized NOP framework in both experiments (*i.e.* of the current paper and the previous one). Such difference demonstrates that the actual implementation over this abstraction layer implemented in C++ causes an overhead in its execution.

In the current materialization of the NOP Framework, in terms of assembly language instructions, causal expressions are certainly more complex and composed of a greater number of instructions to be processed than *if-then* expressions of the OOP. Thus further NOP framework optimizations or new NOP materialization (e.g. NOP compiler and language) would be necessary to achieve NOP potentiality.

6. Conclusions and Future Works

This section discusses NOP features and future works.

6.1. NOP Essence

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

The notification mechanism is composed of entities that collaboratively carry out the inference process by means of notifications, providing solutions to deficiencies of usual paradigms [1,8,9]. In this context, this paper addressed the performance subject making some comparisons of NOP and Imperative/Object Oriented Paradigm instances.

6.2. NOP Performance

In all the experiments generated over the two implementations of the simulator, the results of OOP were better than those of NOP. However, previous asymptotic calculations [10] and other experiments [1] suggest that the efficiency of NOP would be better than OOP, as discussed in [1,10]. Taking into account the current materialization, this would be particularly true to software with large number of causal evaluation and concerned variables with discrete and quite low frequency of changes.

In this paper, a particular fact that negatively impacted in the results was that the simulator be not fully implemented over the NOP principles in the so-called NOP version of Pacman-Simulator. This implementation of the simulator consists of a hybrid version in which only the characters' decisions follow the NOP implementation principles.

Future works should include a more complete version of the simulator based on the NOP principles. However, perhaps even a full NOP version of the Pacmam simulator could produce results without real significant gain with respect to C++ OOP version of the Pacmam simulator, since the current NOP materialization still have the "weight" of a framework over C++. Indeed, NOP should have a language and a compiler to build NOP native code.

Nevertheless, the results of Pacman-Simulator performance in the Optimized NOP Framework (in the current stable release) with respect to Pacman-Simulator in the Original NOP Framework performance were considerable improved. Actually, these results were quite near of the Pacman-Simulator in OOP, as shown in **Table 1**.

Still, the gain of performance between the Optimized NOP Framework and the Original NOP Framework was approximately of 490% over, according to the **Table 1** as well. This fact reinforces that the C++ implementation of the NOP framework strictly impacts in the performance of the applications developed in this manner. Probably, further optimizations of the NOP Framework may provide better results than the current obtained ones, namely in terms of performance.

Besides, actual optimizations are certainly related to the development of a particular compiler and language. Anyway, it would solve some drawbacks of the actual implementation of NOP, such as the overhead of the NOP Framework by using computationally expensive data-structure over an intermediary language (C++). These advances are under consideration in other works.

6.3. NOP Features

In the actual materialization, beyond the so-called Optimized NOP Framework, there is also the so-called NOP Wizard tool to build FBEs, Rules, and other collaborators in high level. Thus, the use of NOP Wizard tends to make easier development of NOP software and save considerable development time with respect to NOP Framework and particularly to OOP. Indeed, the developer would not have to deal directly with the complicated syntax of the OOP and especially with interdependences between causal expressions produced by the imperative approach.

Still, besides NOP supposedly solve redundancy/performance and development problems, it is also potentially applicable to develop parallel/distributed applications because of the decoupling (or minimal coupling, to be precise) of entities. In inference terms, there is no great difference if an entity is notified in the same memory region, in the same computer memory or in the same sub-network.

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.

7. Acknowledgements

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

REFERENCES

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

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

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

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

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

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

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

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

[9] R. F. Banaszewski, P. C. Stadzisz, C. A. Tacla and J. M Simão, "Notification Oriented Paradigm (NOP): A Soft-

ware Development Approach Based on Artificial Intelligence Concepts," *VI Congress of Logic Applied to the Technology—LAPTEC*, Santos, November 21-23, 2007, p. 216.

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

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

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

[13] D. Sevilla, J. M. Garcia and A. Gómez. "Using AOP to Automatically Provide Distribution, Fault Tolerance, and Load Balancing to the CORBA-LC Component Model," *Advances in Parallel Computing*, Vol. 15, 2008, pp. 347-354.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[35] B. D. Wachter, T. Massart and C. Meuter, "dSL: An En-

vironment with Automatic Code Distribution for Industrial Control Systems," *Proceedings of the* 7*th International Conference on Principles of Distributed Systems*, La Martinique, France, 10-13 December 2003, pp. 132-145.

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

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

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

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

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

[41] M. Díaz, D. Garrido, S. Romero, B. Rubio, E. Soler and J. M. Troya, "A Component-Based Nuclear Power Plant Simulator Kernel: Research Articles," *Concurrency and Computation*: *Practice and Experience*, Vol. 19, No. 5, 2007, pp. 593-607. doi:10.1002/cpe.1075

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

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

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

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

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

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

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

[49] D. P. Miranker, "TREAT: A Better Match Algorithm for AI Production System," 6*th National Conference on Artificial Intelligence*—*AAAI*'87, Seattle, 13-17 July 1987, pp. 42-47.

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

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

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

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

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

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

[56] 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.

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

[58] J. Pittman, "The Pac-Man Dossier," 2011. http://home.comcast.net/~jpittman2/pacman/pacmandossier.html