Evaluation of the Notification Oriented Paradigm applied to Real-Time Systems

Robson R. Linhares[1,2,3,5] *, Douglas P. B. Renaux*[1,2,4]*, Jean M. Simão*[1,2,3,4]*, Paulo C. Stadzisz*[1,2,3,5]
1 - Graduate School in Applied Computing (*PPGCA*)
2 - Laboratory of Embedded Systems Innovation and Technology (*LIT-CITEC*)
3 - Graduate School in Electrical Engineering and Industrial Computer Science (*CPGEI*)
4 - Department of Electronic Engineering (*DAELN*)
5 - Department of Informatics (*DAINF)*
Federal University of Technology - Paraná (*UTFPR*) - Curitiba - PR, Brazil
{linhares, douglasrenaux, jeansimao, stadzisz} @ utfpr.edu.br

***Abstract* — Software development based upon current paradigms, such as the Imperative Paradigm (IP) and the Declarative Paradigm (DP), often presents drawbacks such as waste of processing capacity and coupling among entities. This is due to their orientation to a monolithic inference mechanism that is based on causal evaluation implemented by means of searches over passive computational entities. The Notification-Oriented Paradigm (NOP) was conceived as a new approach for conception, structuring, and execution of software leading to performance improvements, organization of causal knowledge, and decoupling of programming entities. The NOP introduces a different manner of structuring software and realization of its inferences, which are based upon small, smart, and decoupled collaborative entities that interact by means of precise notifications. In this way, NOP achieves responsiveness, distributiveness, consistency, and robustness. These features are among the demands of Real-Time Systems. This paper analyzes the NOP applicability to Real-Time Systems by confronting the demands of the latter with the characteristics of the former. As conclusion, the NOP is considered applicable to this sort of computational system.**

Keywords — Real-Time Systems; Notification Oriented Programming.

I. INTRODUCTION

Embedded real-time systems have particular demands for programming. These demands must accommodate the ever increasing number and complexity of requirements as well as the evolutions of the hardware platforms. Nowadays, of particular interest are the multi-core and many-core architectures for which the traditional programming paradigms are becoming less appropriate.

Computing platforms have historically evolved from single-processor to multi-processor architectures [1], either in the form of tightly-coupled multi-core systems or in the form of loosely-coupled distributed systems. Hence, there is an increasing need to efficiently perform distributed computations over multiple cores and multiple network nodes [2][3]. The change to multiprocessor architectures aims at improving *response-time*, *scalability*, *decoupling*, *error isolation*, and *robustness* [1][2][4][5].

Apart from the aforementioned changes in the computing platforms, there is also a change in the manner that the computers are used [6][7][8]. They are progressively more pervasive in different contexts of society, and they are available in a variety of forms and functions such as smartphones, cameras, and GPS. In an era of the Internet of Things (IoT), processing elements are available anyplace, anytime and in any object (anything) [10].

Furthermore, these artifacts of ubiquitous computing are not isolated. Normally, they communicate among themselves or communicate with 'traditional' computing platforms by some means such as wireless technologies [7]. In fact, these collaborative artifacts generate another kind of computing distribution, thereby achieving the ubiquitous computing where an example is the sensor network [8]. In a general way, many of the computing tasks performed by those artifacts (sensor control, communication protocols, etc.) depend on meeting timing requirements and/or constraints to work properly, thus allowing their categorization as real-time computing applications.

The considered context compels the development of new programming techniques in order to facilitate the conception and implementation of the aforementioned real-time computing applications. An example is the Notification Oriented Programming Paradigm (NOP) whose essence is an alternative inference solution based on direct notifications among logical-causal entities and factual entities [3][4][5]. It is believed that the NOP nature could help to develop optimized and distributed software in an easier manner than current approaches [4][5].

There are a number of current research efforts around the NOP. Nevertheless, the efforts in the NOP research about the area of Real-Time Computing are in an early phase, in which it is expected that a number of contributions can be achieved. One of these expected contributions is the development of a NOP language that provides support for real-time systems. Thus, this paper presents an analysis of NOP applicability in the context of Real-Time Computing, in order to evaluate the requirements for this language.

II. NOTIFICATION ORIENTED PARADIGM (NOP)

The main current paradigms can be classified as imperative and declarative ones. The imperative could be established as comprising the procedural and object-oriented approaches, whereas the declarative could be established comprising functional and logic approaches [10]. In short, the imperative

programming presents a lot of code redundancy and coupling, whereas the declarative one presents some coupling and processing overhead in inference solutions because, even with suitable algorithms, they use computationally expensive data-structures [3].

In fact, the current programming paradigm and approaches are driven by monolithic (implicit or explicit) inference, which researches on passive fact base elements (e.g. variables, objects, vectors, etc) in order to test logical/causal expression (e.g. if-then statement or similar rules). This frequently results in code coupling, as well as redundancies or processing overheads as detailed in [3][4]. Therefore, this given context impels efforts to develop new solutions.

A new programming technique, called Notification Oriented Paradigm (NOP), was proposed [4][5]. The NOP basis was initially proposed by J. M. Simão as a manufacturing discrete-control solution [11]. This solution was evolved as a general discrete-control solution [11] and then as a new inference-engine solution [4][5], achieving finally the form of a new programming paradigm [3][4][5].

The NOP presents a new concept to conceive and execute applications based upon notifiable rule-entities composed of collaborative sub-entities. The essence of the NOP is its inference process based upon small, smart, and decoupled sub-entities that collaborate by means of precise notifications [4][5][11]. This solves redundancies and centralization problems of the current approaches of causal-logical processing, thereby solving processing-capacity misuse and coupling issues of the current paradigms [4][5][10].

The idea of the NOP is to make easier the task of building better software, in terms of easier composition of optimized and distributable code [4][5]. In this context, it is expected to save processing resources, thereby enhancing application performance, as well as make it easier to compose multi-core based applications and distributed applications in general.

A. NOP Structural View and Inference Process

In the NOP, the causal expressions are represented by common causal rules, which is natural to programmers of current paradigms and persons in general when a user-friendly interface is used. Anyway, each causal rule is technically dealt with a special computational-entity *Rule*. A *Rule*, in a causal rule form, is illustrated in Figure 1.

Structurally, a *Rule* has a *Condition* and an *Action*, as shown by means of an UML class diagram in Figure 2. Both are entities that work together to carry out the causal knowledge of the *Rule*. The *Condition* concerns to its decisional part related to the referenced element(s), whereas the *Action* concerns to execution related to this(ese) element(s). In the considered example, the referenced element is the *Semaphore1*, which comprises a pair of traffic lights, North-South (NS) and West-East (WE), used to control a crossing in a vehicle traffic control system.

In the NOP, the evaluated elements are represented by an entity type called *Fact_Base_Element* (*FBE*). A *FBE* is composed of one or more attributes. Each attribute is represented by another entity type called *Attribute*, for example *queuedCars_NS* and *TrafLightNS_State* in the case of the *FBE Semaphore1*. The states of *Attributes* are analyzable, in an inference process, in the *Conditions* of *Rules* by using other collaborator entities called *Premises* as modeled in Figure 2. In the considered *Rule* (Figure 1), the *Condition* is composed of three *Premises*.

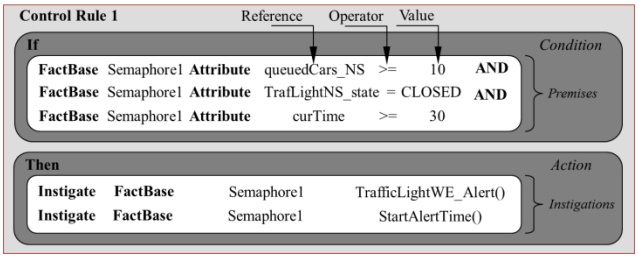

Figure 1 - The representation of a Rule.

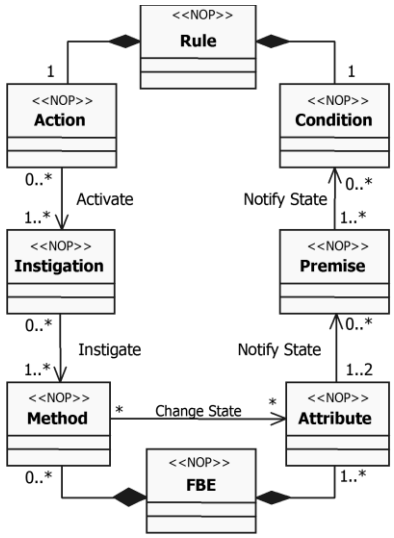

Figure 2 – Rule and Fact_Base_Element class diagram [4][5]

When each *Premise* of a *Condition* is inferred as true, the *Rule* becomes true and may activate its *Action* composed of entities called *Instigations*. In the considered *Rule*, the *Action* contains two *Instigations*. In fact, *Instigations* are linked to and instigate *Methods* (*TrafficLightWE_Alert* and *StartAlertTime* in the considered *Rule*), which are another entity of *FBE*. Each *Method* allows executing *FBE* services. Generally, the call of *FBE Method* changes *FBE Attribute* states, feeding the inference process.

The inference process of the NOP is innovative once the *Rules* have their inference carried out by active collaboration of its notifier entities [4]. In short, the collaboration happens in the following way: 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 among the collaborators.

In order to detail the inference process by notification, it is firstly necessary to explain the *Premise* nature and composition. Each *Premise* represents a Boolean value about

one or even two *Attribute* states and is composed of: (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 value of other referenced *Attribute* also received by notification.

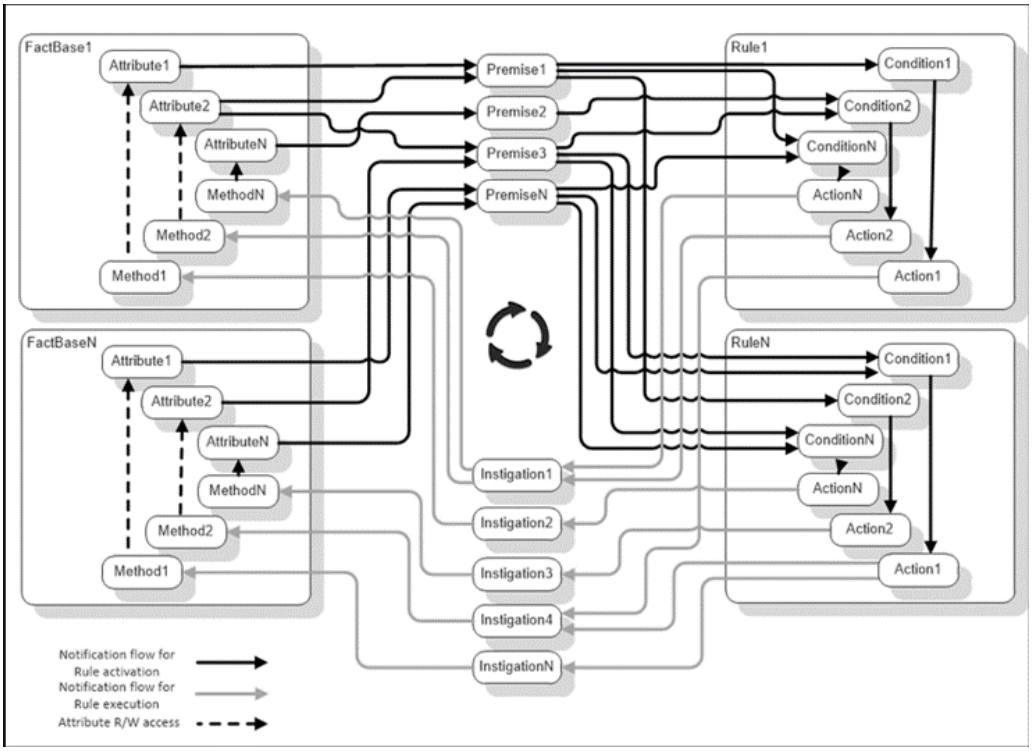

Figure 3 – Rule Notification chain [12]

A *Premise* makes a logical calculation when it receives notification of one or even two *Attributes* (*Reference* and even *Value*). This calculation is carried out by comparing *Reference* and *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 *Conditions*. Thus, each notified *Condition* calculates its Boolean value by the conjunction of the *Premises* values.

When all *Premises* that integrate a *Condition* are satisfied, the *Condition* itself is satisfied and notifies the respective *Rule* to execute. The collaboration among the NOP entities by means of notifications can be observed at the schema illustrated in Figure 3. In this schema, the flow of notifications is represented by arrows linked to rectangles that, in turn, represent the NOP entities.

An important point to clarify about the collaborative entities of the NOP is that each notifier (e.g. an *Attribute*) registers its clients (e.g. *Premises*) in its 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 elements to be notified when its state change. Of course, all entities composition and links should be done in friendly environment.

B. NOP Nature

In NOP, each *Attribute* state is evaluated by a set of logical and causal expressions (i.e. *Premises* and *Conditions*) in the changing of its state. Thanks to the cooperation by means of precise notifications, the NOP avoids the two types of redundancies verified inclusively in imperative language, the temporal redundancy (unnecessary logical/causal expression evaluation) and structural redundancy (repetition of logical expression in causal ones).

The temporal redundancy is solved in the NOP by eliminating searches over passive elements, once some data-entities (i.e. *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 (i.e. *Premises*), avoiding that other parts and even other causal expressions be unnecessarily evaluated or re-evaluated.

Also, the structural redundancy is solved in NOP when a *Premise* 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. Indeed, the avoidance of structural redundancies and mainly the avoidance of temporal ones allow improving performance [4].

Besides solving performance problems, the 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 not 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 [4]. However, these issues are under technical implementation and experimentation.

Actually, it would be needed to implement particular language and compiler to NOP, which could eventually take into account the ease of distribution. Moreover, this technology could improve performance by technically optimizing the implementation of data-structures of NOP entities. This technology is currently under development. Meanwhile, in order to allow implementations based on the NOP, its entities were materialized in the form of classes in a C++ framework that are instantiated by developed applications [10]. Moreover, a wizard tool has been proposed to automate and thus facilitate this process. It is the NOP Development Environment – a tool that generates the NOP smart-entities from causal rules elaborated in a graphical interface.

In this case, software developers "only" need to implement *FBEs* with *Attributes* and *Methods*, once other NOP special-entities will be completely composed and linked by the tool. This allows using the time mainly to the construction of the causal base (i.e. in the composition of the NOP rules) without concerns about the instantiation of the NOP entities.

III. Requirements for Real-Time Programming Languages

A programming language that is used in the development of a Real-Time System should provide specific support for this kind of system. This support includes [13][14]:

- **Time management** - including access to a real-time clock, task suspension for a specified time (delay) and the specification of timeouts.
- **Deadlines and Scheduling** - the programmer should be able to explicit the deadline of each job as well as specify the scheduling policy and scheduling related characteristics of tasks, such as the task priority.
- **Support for Schedulability Analysis** - languages constructs that specify an upper limit to loop iterations provide relevant input for schedulability analysis
- **Concurrency** - a real-time programming language must provide constructs that explicit the concurrency of the application, as well as inter-task communication and synchronization
- **Dependability** - a language may provide partial support for dependability via strong type checking and exception handling

The C Programming Language provides none of the requirements listed above. Not even its type checking is strong enough. This does not preclude its use for programming real-time systems. In fact, it is the most used language for embedded and real-time systems [15]. C programmers have to rely on RTOS services to provide some of these requirements. Languages such as Ada provide some support for these requirements. Nevertheless, real-time support in NOP will be evaluated against these requirements.

IV. Notification Oriented Paradigm (NOP) Properties

This current section presents properties of the Notification Oriented Paradigm (NOP) and considerations about its suitability to Real-Time Computing, in terms of the desired Real-Time programming requirements presented in the previous section.

A. Time management

NOP software is able to instantiate a set of *Attributes* to be used as software timers. These software timers would be suitable to control task suspension and generate timeouts for the real-time tasks.

In fact, one *Attribute* can be instantiated and configured to have its value decremented by an interrupt service routine (ISR) that is triggered by a hardware timer. This *Attribute* would notify a concerned *Premise*, which would evaluate if the *Attribute* initial value has reached zero and, if so, would generate a notification to the concerned timer *Condition*. This same mechanism can be used to provide access to the Real-Time clock value, as long as its ISR can also be configured to update a set of NOP *Attributes.*

As the notification from a certain timer to its related *Condition* is only generated when the timer expires, the temporal redundancies related to repeatedly testing the timer expiration are avoided. This would lead to a more efficient use of processing capacity, which is a characteristic of NOP that is useful, particularly when dealing with a set of tasks that have real-time requirements.

The responsiveness to the timers can be set by programming the relative priorities of the involved *Rules* (i. e. the *Rules* that can be activated by the timer-related *Conditions*). The responsiveness of NOP software in general has been discussed in previous work [12], in terms of computational time complexity when compared to other paradigms.

Regarding the requirements for a Real-Time NOP Language, they include time management primitives, in the form of specific annotations, to properly categorize some *Attributes* as software timers.

B. Deadlines and scheduling

The activation of a set of NOP *Rules*, and consequently the activation of their connected FBE *Methods*, is what essentially triggers the execution of useful work in NOP software. Therefore, a single *Rule* (or even every one of its *Methods*) can be considered a "task" in this context. Additionally, the NOP elements that execute logical-causal evaluations (*Premises* and *Conditions*) can also be considered "micro-tasks" (i.e. tasks with a very small granularity), in the sense that they also have to be scheduled for execution upon reception of a notification. The fine granularity of each schedulable NOP element ("micro-task') improves the flexibility of the allocation activity, as each "micro-task" depends solely on a communication channel to receive/propagate notifications from/to other micro-tasks.

The scheduler can treat each of these "micro-tasks" as an independently-schedulable unit, given the low coupling among the corresponding NOP elements and given their control flow that is only dependent on the propagation of the notifications. This motivates the definition, in the Real-Time NOP Language, of timing annotation constructs for each of the NOP elements, which could be used by the programmer to provided relevant timing information. These annotations could be dynamically used by a scheduler to organize the ready queue based on policies such as earliest deadline first, for instance. For this purpose, timing annotations of "micro-tasks" could be complemented by annotations concerning the notification path for end-to-end "macro-tasks" (e. g., from sensing an external attribute to effectively actuating over the environment).

It is worth noticing that, despite the fine granularity of each schedulable element, the notification dynamics helps reducing the scheduling overhead because each "micro-task" is only eligible for scheduling as it receives a notification. As every "micro-task" is simple and inherently sequential, its temporal analysis would not incur in determining loop boundaries or infeasible paths. Hence, its timing annotations can easily be automatically generated by a static timing analysis tool.

With respect to the scheduling priority, it can be supported by means of the relative priorities of a set of activated *Rules.*

C. Support for schedulability analysis

The inference method performed in NOP applications is closer to the Petri nets' nature than those inference methods based on search, since Petri nets somehow operate like notifications [4]. These aspects were particularly detailed in [11] in terms of control solution, in [4] in terms of inference solution, and in [16] in terms of development paradigm. The implication of this characteristic is that *Rules* can be automatically and quite directly translated to Petri nets and, possibly, to timed Petri nets by adding the suitable timing annotations to the model.

Since the Real-Time NOP Language shall provide primitives for timing annotations, the schedulability analysis can make use of such timing annotations regarding each "micro-task" and also regarding the expected frequency of notifications among NOP elements. These annotations can be provided by the user or inferred through static analysis of the NOP software. As the timed Petri nets have been successfully used to model real-time systems [17], they could be used as a suitable approach to obtain the timing information and expected frequency of notifications among NOP elements, thus providing support for schedulability analysis.

D. Concurrency

The concurrency is explicit in NOP software, given that the execution of a logical-causal relation defined by a *Premise* or a *Condition* or the execution of a *Method* can be potentially triggered (by notification) simultaneously to the execution of other NOP elements.

Inter-task communication is intrinsically implemented in NOP software by means of *Methods* that update *Attributes*. This is due to the fact that updating *Attributes*, by a certain *Rule* (task) and its *Methods*, is the primary mechanism to trigger a new cycle of notifications that would eventually lead to the execution of other *Rules* that represent other (micro) tasks.

Synchronization mechanisms, such as semaphores and mutexes, can be conceptually implemented in NOP software. However, in order to allow the scheduling of the "micro-tasks", the propagation of notifications among NOP elements would eventually not be atomic, which could lead to inconsistencies due to race conditions. To avoid this, it would be necessary to improve the behavior of NOP elements (such as *Methods*) to allow atomic operations in the form of "test and set" that would be suitable to implement safe synchronization mechanisms, or even implement mechanisms based on transactional memories. The mechanism based on "test and set" is already implemented in a computer architecture that is specifically designed to execute NOP software [18]. It can be used by the Real-Time NOP Language to define critical regions, i. e., sequences of enchained *Methods* that must be executed atomically.

It is important to emphasize that a single NOP *Method* can implement a sequence of operations, similar to a function in imperative programming, which would facilitate implementations of semaphores or mutexes according to classical algorithms such as Peterson´s [19]. However, this sort of *Method* would not be schedulable as a "micro-task", thus relying on schedulability analysis techniques similar to those used for imperative programming.

E. Dependability

As previously discussed in [12], in the context of Sentient Computing, the NOP software can make use of some mechanisms to improve robustness. These mechanisms include techniques to achieve deterministic inference and to create synchronized NOP elements that can operate redundantly. The NOP language can also provide constructs that support the explicit use of those mechanisms.

The improved robustness of NOP, mainly in terms of redundancy of operation, helps improving the availability and reliability of the NOP applications. The availability and the reliability are two of the main attributes that should be maximized in order to improve the dependability of a system [20].

Also, the maintainability attribute of a dependable system can be favored by the fact that the NOP elements (including *Rules*) are highly decoupled, thus facilitating maintenance operations. It includes not only corrective and preventive maintenance, which could be achieved by activation or deactivation of existing *Rules* and their related NOP elements, but also adaptive and augmentive maintenance as proposed by [20] that would involve the creation and addition of new *Rules* to the software.

F. Table of Real-time programming requirements x NOP properties

For the sake of succinctness, Table I presents the set of requirements concerning real-time programming here highlighted and how they are facilitated or fulfilled by the NOP.

TABLE I. REAL-TIME PROGRAMMING X NOP

N	***Requirement***	***NOP***
A	Time management	Yes (specific *Attributes* for software timers)
B	Deadlines and scheduling	Partially (micro tasks)
C	Support for schedulability analysis	Yes (timing annotations for micro tasks)
D	Concurrency	Yes
E	Dependability	Yes

V. CONCLUSION AND FUTURE WORK

NOP is characterized by its problem representation in the form of causal rules, which are composed of elements that can be considered as "micro-tasks" with fine granularity and high degree of parallelism and decoupling. These characteristics, together with suitable timing annotations, tend to facilitate the scheduling and timing analysis of the NOP software.

The NOP elements also tend to be very efficient on using processing resources, because they activate and effectively execute some processing only upon reception of notifications from other elements. Additionally, NOP software can make use of determinism and redundancy mechanisms that are being developed in the context of the paradigm.

The combination of these characteristics and their effects (i. e., improved efficiency, robustness and ease of schedulability analysis) completely or partially meets the requirements for real-time programming as reviewed in this paper. Thus, the preliminary analysis presented here indicates a high applicability of NOP to Real-Time programming.

This research will proceed by implementing the Real-Time NOP Language including its timing annotations, as well as the development of timing analysis techniques and tools to support the schedulability analysis of NOP software. These investigations and their results will support a more detailed analysis of the applicability of NOP to real-time software programming.

REFERENCES

[1] W. A. Gruver. "Distributed Intelligence Systems: A New Paradigm for Systems Integration". IEEE International Conference on Information Reuse and Integration, 2007.

[2] G. Coulouris, J. Dollimore, and T. Kindberg. "Distributed Systems – Concepts and Designs". Pearson – Addison Wesley, 2001.

[3] J. M. Simão, R. F. Banaszewski, C. A. Tacla, P. C. Stadzisz, "Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study," Journal of Software Engineering and Applications (JSEA), p.402-416, v.5, n.6, 2012.

[4] J. M. Simão and P. C. Stadzisz, "Inference Based on Notifications: A Holonic Meta-Model Applied to Control Issues". IEEE Transactions on Systems, Man and Cybernetics, Part A. Vol. 39, Issue 1, Jan. 2009 Pg. 238-250 (Submitted on July 2007, Approved on July 2008). Digital Object Identifier 10.1109/TSMCA.2008.2006371.

[5] J. M. Simão, P. C. Stadzisz, "Notification Oriented Paradigm (NOP) — A Notification Oriented Technique to Software Composition and Execution". Original title in Portuguese "Paradigma Orientado a Notificações (PON)—Uma Técnica de Composição e Execução de Software Orientada a Notificações". Patent pending submitted to INPI/Brazil in 2008 and UTFPR Innovation Agency 2007. INPI Number: PI0805518-1. http://www.patentesonline.com.br/paradigma-orientado-a-notificacoes-pon-uma-tecnica-de-composicao-e-execucao-de-software-234943.html.

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

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

[8] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. "Wireless sensor networks: a survey". Comput. Networks, 38(4): 393-422, 2002.

[9] International Telecommunication Union. "ITU Internet Report 2005: The Internet of Things." (2005).

[10] R. F. Banaszewski, "Notification Oriented Paradigm: Advances and Comparisons". Original title in Portuguese: "Paradigma Orientado a Notificações: Avanços e Comparações". Master in Science Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology - Paraná (UTFPR). Curitiba, Paraná, Brazil, March 27, 2009. http://arquivos.cpgei.ct.utfpr.edu.br/Ano_2009/dissertacoes/Dissertacao_500_2009.pdf .

[11] 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 – Double Diploma – Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at The Federal University in Technology of Paraná (CEFET-PR/UTFPR), Brazil – Research Center for Automatic Control of Nancy (CRAN), Henry Poincaré University (UHP), France, June, 2005 (at CPGEI/UTFPR) – http://tel.archives-ouvertes.fr/docs/00/08/30/42/PDF/ThesisJeanMSimaoBrazil.pdf.

[12] J. M. Simão, D. P. B. Renaux, R. R. Linhares, and P. C. Stadzisz, "Evaluation of the Notification Oriented Paradigm applied to Sentient Computing". In Proceedings of the 17th International Symposium of Object/Component-Oriented Real-Time Distributed Computing (ISORC 2014), p. 253-260, Reno, USA, June 2014.

[13] A. Burns and A. J. Wellings. "Real-Time Systems and Programming Language". Addison Wesley, 2nd edition, 1996.

[14] W. A. Halang and K. Mangold. "Real-Time programming languages". In M. Schiebe and S. Pferrer, editors, Real-Time Systems Engineering and Applications, chapter 6, pages 141-200. Kluwer Academic Publisher, 1992.

[15] D. Blaza and A. Wolfe, "2013 Embedded Market Study," DesignWest, April 2013. Available at: http://e.ubmelectronics.com/2013EmbeddedStudy/index.html

[16] J. M. Simão, P. C. Stadzisz, L. V. B. Wiecheteck. "UML Profile to Notification Oriented Paradigm (NOP), UML Profile to Rule Oriented Paradigm (ROP), Notification Oriented Development (NOD) Method, and Rule Oriented Development (ROD) Method". Original title in Portuguese: "Perfil UML para o Paradigma Orientado a Notificações (PON), Perfil UML para o Paradigma Orientado a Regras (POR), Método de Desenvolvimento Orientado a Notificações (DON) e Método de Desenvolvimento Orientado a Regras (DOR)". 2012. Patent pending submitted to INPI/Brazil in 2012 and UTFPR Innovation Agency 2012. INPI Provisory Number: BR 10 2012 026430 7

[17] A. Cerone and A. Maggiolio-Schettini, "Time-based expressivity of timed Petri nets for system specification". Theoretical Computer Science, Vol. 216 Issue 1-2, p. 1-53, 1999. DOI 10.1016/S0304-3975(98)00008-5.

[18] R. R. Linhares, "A Contribution to the Development of a Computer Architecture Proper to the Notification Oriented Paradigm". Original title in Portuguese: "Contribuição para o Desenvolvimento de uma Arquitetura de Computação Própria ao Paradigma Orientado a Notificações". Doctorate Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology - Paraná (UTFPR). Curitiba, Paraná, Brazil, 2014. Unpublished work.

[19] G. L. Peterson, "Myths About the Mutual Exclusion Problem". Information Processing Letters 12(3), p. 115-116, 1981.

[20] A. Avizienis, J-C. Laprie, B. Randell and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing". IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, p. 11-33, 2004.