Application of generic programming for the development of a C++ framework for the Notification Oriented Paradigm

Felipe dos Santos Neves*‡, Jean Marcelo Simão†§, Robson Ribeiro Linhares*¶,
*Informatics Department (DAINF)
†Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI)
Federal University of Technology - Paraná (UTFPR) - Curitiba - PR, Brazil
‡fneves@alunos.utfpr.edu.br, §jeansimao@utfpr.edu.br, ¶robson@dainf.ct.utfpr.edu.br

***Abstract*—Modern software development mainly relies on the use of the Imperative Paradigm (IP) and Declarative Paradigm (DP). However, despite widespread use of IP/DP, they have their drawbacks, such as the presence of structural and temporal redundancy, as well as cod coupling. As an alternative, the Notification Oriented Paradigm (NOP) introduces a new approach for developing software, based on the use of small reactive notifiable entities. Therefore, the NOP facilitates software development and allows achieving features such as responsiveness by redundancy avoidance and distributiveness by code decoupling. Some frameworks have been implemented to allow the development of NOP software applications in various programming languages, thereby changing the usual *modus operandi* of these languages. However, those NOP frameworks present some shortcomings, such as type limitation. This paper evaluates such shortcomings and proposes a new framework version based on generic programming, called NOP C++ Framework 4.0, showing how it improves in performance and usability over the most stable existing framework, the NOP C++ Framework 2.0.**

***Index Terms*—Notification Oriented Paradigm, Framework, Generic Programming, C++17**

I. Introduction

Programming paradigms can be defined as models or methods for developing software, according to a set of principles, such as linguistic and mathematic ones. These paradigms are defined to aid to solve a given set of problems according to their models and principles. Each paradigm, with its own set of rules, can be better or worse in solving a specific type of problem [1].

In modern computer system development, among the existing programming paradigms, the Imperative Paradigm (IP) and the Declarative Paradigm (DP) are the main ones, as they are the most well-known and used. However, these paradigms still present some problems due to the sequential loop/search-oriented execution model. Actually, DP is a sort of loop automatization of IP loops by the so-called inference machine [2].

In short, due to its loop orientation, the IP introduces temporal and structural redundancy in the code, as well as code-piece coupling, thereby generating processing overhead and complicating activities such as code reuse and parallelism and/or distribution. Despite its different approach by automating loops by its inference machine, the DP still presents similar problems, namely overhead processing and code-piece coupling. This happens due to its inference mechanisms based on search over passive elements [2].

Aiming to solve these problems, the Notification Oriented Paradigm (NOP) is introduced as a new model for the development of computational systems. Section II describes the NOP in greater detail. The current state of development of the NOP in software is also presented therein, in which the NOP C++ Framework 2.0 is highlighted as the most stable and mature implementation. The shortcomings of this implementation, with respect to the difficulties to its use, are explained as well.

In this context, this paper introduces a new proposed version of the framework called NOP C++ Framework 4.0. Section IV elaborates over the main concepts utilized for the development of the NOP C++ Framework 4.0, highlighting how these concepts may improve the NOP usability of the framework, as well as presents performance comparisons between the NOP C++ Framework 2.0 and NOP C++ Framework 4.0 and their amount of code.

II. Notification Oriented Paradigm (NOP)

The NOP is originated from a manufacturing discrete control solution [4], which later evolved to a general discrete control solution, alternative inference solution, and finally to a new programming paradigm [5].

The NOP programs can be and usually are declaratively expressed as presented in Fig. 1. It represents the fact-executional (object-like) and logic-casual (rule) entities for an application of sensor correlation. This figure shows a rule for a sensor application. This sensor is represented as a Fact Base Element (FBE) composed of two boolean *Attributes*, *atIsRead* and *atIsActivated*, and a *Method mtProcess* which sets the values of *atIsActivated* to *false* and *atIsRead* to *true*. That *Rule* is composed of a *Condition* and an *Action*. This *Condition* is approved when both its *Premises* are *true*, executing the *Rule*, which causes the *Action's Instigation* to instigate the *Method mtProcess* from the sensor *FBE*.

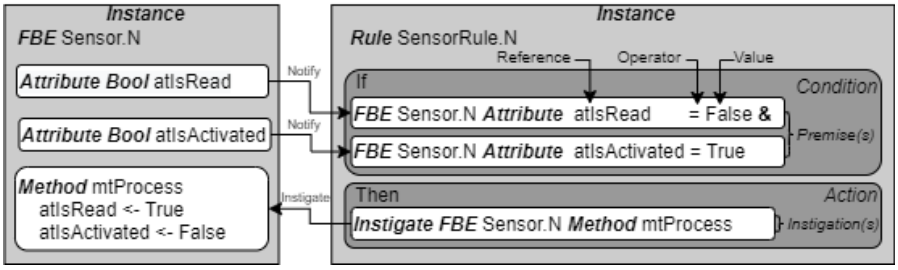

Fig. 1: Example of NOP *Rule* for a sensor application

The main feature of the NOP is the notification mechanism. Each element of the NOP meta-model can send and/or receive precise and pertinent notifications, thereby allowing the evaluation of states only when a notification arrives. This presents a new way to conceive and execute applications based on small reactive notifiable entities, which avoids temporal/structural redundancies and promotes code-piece fine-grained decoupling. Thus, among others, the NOP facilitates the development of performing and distributable code [5].

The NOP meta-model entities described above by means of an example are, more precisely, described below:

- *Fact Base Element* (*FBE*): Entity that contains the *Attributes* and *Methods*.
- *Attribute*: Entity responsible for storing a value that represents property states of an *FBE*; an *Attribute* differs from a normal variable by being able to notify its concerned *Premises* when its value changes.
- *Premise*: Entity that, when notified by an *Attribute*, performs a logic-relational comparison (i.e., equal, different, greater or equal, less or equal, greater, less) between this *Attribute* with a constant or other *Attribute*, thereby resulting in a logical value. It notifies concerned *Conditions* when its logical value changes.
- *Condition*: Entity that groups one or more *Premises* and, when notified by one of the *Premises* from which it depends, performs a logical operation over their approval states (i.e., normally a conjunction or a disjunction). It notifies its concerned *Rule* when its logical value changes to *true*.
- *Rule*: Entity that has a *Condition* entity and an *Action* entity. Each *Rule* determines the execution of its *Action* when its *Condition* is approved (i.e., state as *true*).
- *Action*: Entity that is attached to a *Rule*. When an *Action* is notified by its *Rule*, it activates its own *Instigations*.
- *Instigation*: Entity that is related to one or more *Actions*. Each *Instigation* entity instigates a set of *Method* entities when notified by an *Action*.
- *Method*: Entity that executes a function or service of an *FBE* when instigated by an *Instigation*. After its execution, each *Method* may change the states of *Attributes*, thereby feeding a notification inference cycle.

As observed by the above NOP entity description, the NOP divides the execution of software into two groups: logic causal and fact executional. In the logic causal group, the entities collaborate in the process of evaluating states to approve or disapprove a *Rule*, whereas the fact executional group is composed of the steps required to execute the *Methods* that may cause changes in the *Attribute*'s values, therefore providing feedback to the notification inference chain [2] [3].

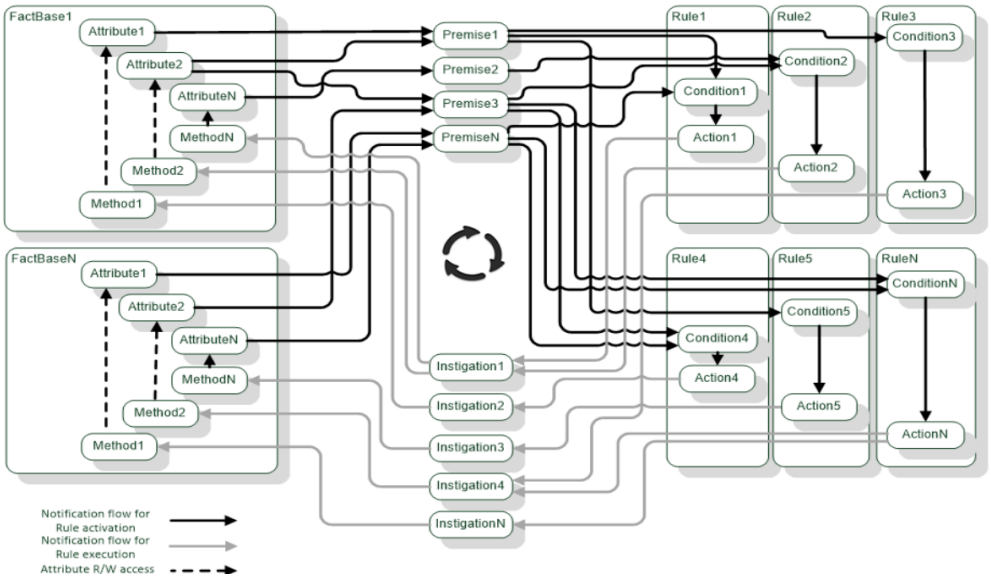

Fig. 2: Representation of the NOP entity notification chain [3]

This collaboration between entities through notifications is illustrated by the schema in Fig. 2, which is actually an instance diagram of the NOP entities

Multiple efforts have been done in order to enable developers to conceive and implement both software and even hardware in NOP. This paper is particularly focused on NOP software materializations, namely framework ones, which are presented in the following section.

III. NOP SOFTWARE MATERIALIZATIONS

The NOP presents multiple materializations in software, through frameworks in different programming languages, as well as its own programming language.

There are four different framework versions in C++: the NOP C++ Framework Prototypal [5], NOP C++ Framework 1.0, NOP C++ Framework 2.0 [2] and NOP C++ Framework 3.0 [6]. The NOP C++ Framework Prototypal was the first materialization in order to test NOP viability. It was followed by the NOP C++ Framework 1.0, whose proposal was to facilitate the composition of software using NOP.

Afterward, the NOP C++ Framework 2.0 was developed focusing on performance [7] [8]. It introduced the application of design patterns and proper data structures, becoming the main framework for the development of NOP applications, due to its maturity and stability [9]. Fig. 3 shows the framework core class diagram, in which it is possible to see the relations of the NOP entities implemented as classes, as well as the derived classes in Fig. 4, which are used to specialize *Attributes* and *Conditions*.

Finally, there is also the NOP C++ Framework 3.0, which proposed to extend NOP C++ Framework 2.0 for use in multi-threaded environments. However, the performance benefits of this version were not as expected, as the parallel execution control mechanisms ended up increasing the processing times. Code 1 shows the implementation of the sensor *FBE* and *Rule* from Fig. 1 using the NOP C++ Framework 2.0.

```
struct NOP2Sensor : public FBE
{
  Boolean* atIsActivated, atIsRead;
  Premise* prIsActivated, prIsNotRead;
  Instigation* inSensor;
  RuleObject* rlSensor;
  Method* mtSensor;
  NOP2Sensor()
  {
    BOOLEAN(this, atIsActivated, false);
    BOOLEAN(this, atIsRead, false);
    mtSensor = new MethodPointer<NOP2Sensor>(this, &NOP2Sensor::ProcessSensor);
  }
  void ProcessSensor() { atIsRead->setValue(true); atIsActivated->setValue(false); }
};
class SensorApp : public NOPApplication
{
public:
  SensorApp(int count) : NOPApplication()
  {
    SingletonFactory::changeStructure(SingletonFactory::NOPVECTOR);
    SingletonScheduler::changeScheduler(SchedulerStrategy::NO_ONE);
    for (int i = 0; i < count; i++)
    {
      sensors.push_back(std::make_shared<NOP2Sensor>());
    }
    for (auto& sensor : sensors) {
      PREMISE(sensor->prIsActivated, sensor->atIsActivated, true,
              Premise::EQUAL, Premise::STANDARD, false);
      PREMISE(sensor->prIsNotRead, sensor->atIsRead, false,
              Premise::EQUAL, Premise::STANDARD, false);
      INSTIGATION_NAMED(sensor->inSensor, sensor->mtSensor);
      RULE(sensor->rlSensor, SingletonScheduler::getInstance(),
           Condition::CONJUNCTION);
      sensor->rlSensor->addPremise(sensor->prIsActivated);
      sensor->rlSensor->addPremise(sensor->prIsNotRead);
      sensor->rlSensor->addInstigation(sensor->inSensor);
      sensor->rlSensor->end();
    }
  }
  std::vector<std::shared_ptr<NOP2Sensor>> sensors;
};
```

Code 1: Code for the sensor structure using NOP C++ Framework 2.0

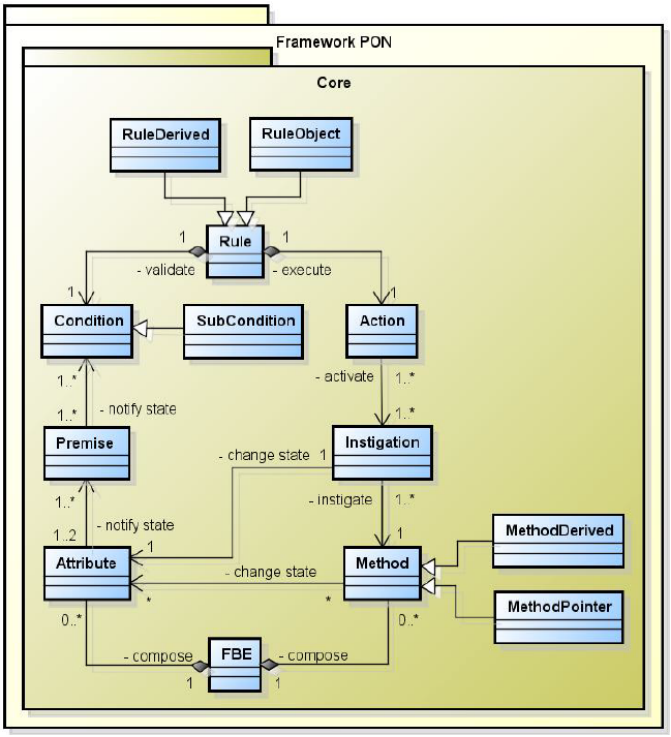

Fig. 3: Core class diagram of the NOP C++ Framework 2.0 [9]

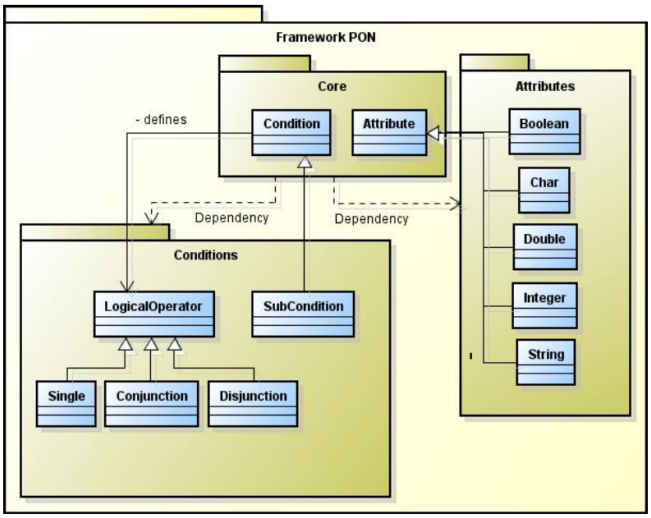

Fig. 4: Extension of the class diagram of the NOP C++ Framework 2.0 [9]

Furthermore, there are versions developed in other programming languages. There are the NOP Java Framework 1.0 and the NOP C# Framework 1.0, which were adaptations of the NOP Framework C++ 1.0. There is also the C# NOP Framework IoT, an adaptation of the NOP C# Framework 1.0, which proposed the distribution of the NOP entities, allowing the application of IoT concepts [10]. Finally, there are the NOP Framework Elixir/Erlang [11], as well as the NOP Framework Akka.NET, both using the actor model in their implementation for multi-core applications, based on the inherent distribution properties of the NOP. In general, the NOP frameworks are still not as performing as the NOP theory preconizes, being NOP C++ Framework 2.0 the one with more

acceptable response times even if not ideal.

Besides, there is also the Notification Oriented Paradigm Language (NOPL) technology, which introduces a method for the creation of languages and compiler for the NOP. This allows the development of NOP software in a high-level rule-oriented language that can be compiled to other target, such as the frameworks introduced above or other dedicated specific notifying code in programming languages to achieve better performance [9] [12].

Even though the NOPL allows the development of high-level code in NOP, it is still difficult to develop complete application with it, since the NOP technology lacks the support of basic functionality such as networking, user I/O interfaces, and file handling. Actually, the NOPL technology is very new and is under development and tests, whose maturity will be achieved just in some years ahead.

In that sense, the NOP frameworks are still the easiest way to develop an actual application using NOP. However, the NOP frameworks have some shortcomings, inclusively the above highlighted NOP Framework C++ 2.0. Even if it is the main framework for the development of NOP application there are shortcomings, whose the main three ones are pointed out below:

- Verbosity: The framework demands a large amount of code to be written in order to achieve even basic functionality, increasing the difficulty in its use. This is due to the object creation interfaces, which sometimes also require writing multiple lines of boilerplate code.
- Low type flexibility: The framework only supports a limited set of basic data types (int, bool, double, and string), thereby not allowing custom user types and then making it difficult to interface with external code. This is due to the project choice for the framework to create dedicated data structures to handle each type.
- Low algorithmic flexibility: The framework uses a strict structure for the *Premises* and *Conditions* that do not allow the user to create more complex evaluations with ease, requiring the creation of many auxiliary objects to achieve the desired functionality. The *Condition* only allows for the evaluation of conjunction or disjunction of *Premises*, instead of allowing more complex logic-causal expressions.

IV. NOP C++ Framework 4.0

Given the shortcomings of the current implementations of the NOP frameworks, the development of the NOP C++ Framework 4.0 is proposed. The goal of the NOP C++ Framework 4.0 is to create a new framework that improves upon the already existing NOP C++ Framework 2.0 and solves the shortcomings mentioned in Section II.

For this purpose, modern C++ development techniques are utilized, such as generic programming. Generic programming consists of writing code that is able to support different types of data, where the type is given as an argument that is interpreted by the compiler [13]. Writing code with generic programming does not incur performance losses, since the translation to static types is done by the compiler, thus there is no execution cost [14].

To make better use of the generic programming techniques available in C++, the C++ 17 standard is chosen to be used in this project, as it contains useful functionalities for writing generic code, such as variadic templates and fold expressions. The class diagram of the NOP C++ Framework 4.0 is shown in Fig. 5. It highlights the use of template classes for supporting *Attributes* that store values with generic types and the reflexive association in the *Condition* class, which allows for a more flexible algorithmic structure. This class diagram is very similar to the class diagram of the NOP C++ Framework 2.0 in Fig. 3, however, it does not need to use specialized classes such as the ones in Fig. 4 due to their generic implementation.

Besides the usage of generic programming for the base class structure, it is also used to enable a flexible object creation interface, through the builder pattern, in which variadic templates and fold expressions are used in order to create builder functions that allow for a variable number of parameters.

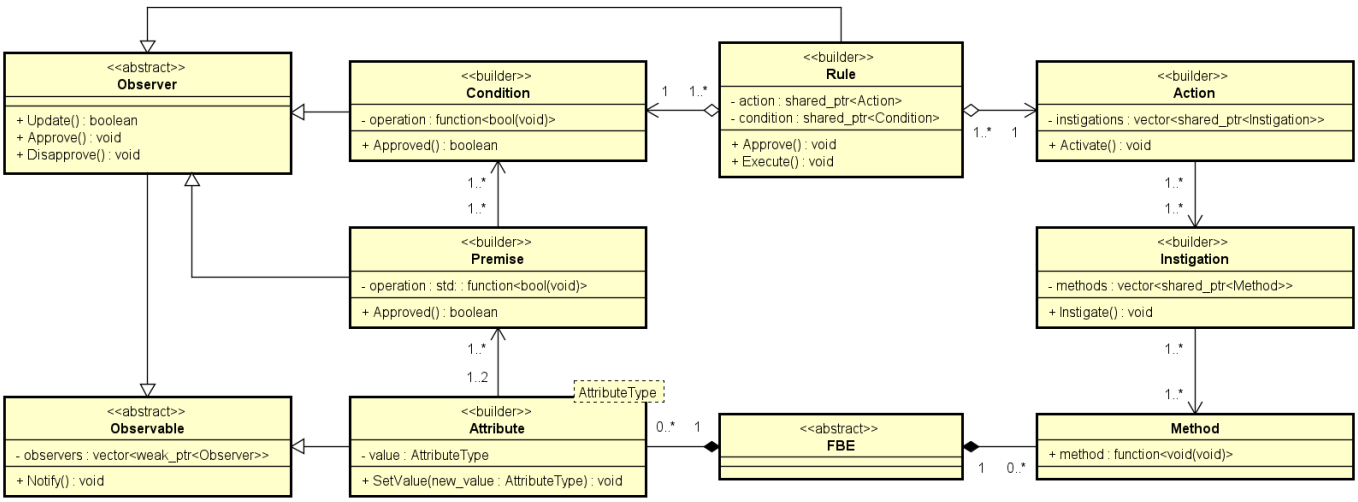

Fig. 5: Class diagram of the NOP C++ Framework 4.0

Finally, the usage of lambda expressions for the evaluation of the state of *Premises* and *Conditions* allows the creation of a generic function that can support any comparison. In the case of the *Condition*, it allows representing the knowledge more easily by allowing the user to write the *Condition* as a generic logic-causal expression, using both *Premises* and other *Conditions* in its composition.

The Code 2 is shown as an example of the usage of all these features combined to create sensor structure used to implement the *Rule* shown in Fig. 1 for the sensor application.

```
struct NOPSensor{
   NOP::SharedAttribute<bool> atIsRead{ NOP::BuildAttribute(false) };
   NOP::SharedAttribute<bool> atIsActivated{ NOP::BuildAttribute(false) };
   NOP::SharedPremiseprIsActivated{ NOP::BuildPremise<bool>(atIsActivated, true,
         NOP::Equal()) };
   NOP::SharedPremiseprIsNotRead{ NOP::BuildPremise<bool>(atIsRead, false,
         NOP::Equal()) };
   NOP::SharedRulerlSensor{ NOP::BuildRule(
      NOP::BuildCondition<NOP::Conjunction>(prIsActivated, prIsNotRead),
      NOP::BuildAction(NOP::BuildInstigation([&]() {
        this->Read();
        this->Deactivate();
        })))
   };
   void Read() const { atIsRead->SetValue(true); }
   void Activate() const { atIsActivated->SetValue(true);
         atIsRead->SetValue(false); }
   void Deactivate() const { atIsActivated->SetValue(false); }
};
```

Code 2: Code for the sensor structure using NOP C++ Framework 4.0

V. Results

With the implementation of the NOP C++ Framework 4.0 presented in Section IV, some comparisons can be drawn with the NOP C++ Framework 2.0. The NOP C++ Framework 2.0 requires specific code to handle each type of data, resulting in a larger codebase, composed of multiple instances of duplicated and redundant code. The NOP C++ Framework 2.0 has 6587 lines of code composing the framework, and NOP C++ Framework 4.0 has only 472 lines.

A performance comparison shows that the NOP C++ Framework 4.0 allows faster execution when compared with the NOP C++ Framework 2.0 best practice. Furthermore, it is noticeable that utilizing the NOP C++ Framework 4.0 resulted in less code to implement an equivalent structure for this example. Code 3 shows the representation with NOPL for the sensor FBE. The NOPL code can be used to generate code for the frameworks, but, for the purpose of optimization, the code was written manually for both frameworks in this test.

```
fbe Sensor
  public boolean atIsRead = false
  public boolean atIsActivated = false
  private method mtProcess
    attribution
      this.atIsRead = true
      this.atIsActivated = false
    end_attribution
  end_method
  rule rlSensor
    condition
       premise prIsActivated
         this.atIsActivated == true
       end_premise
       and
       premise prDebug
         this.prIsNotRead == false
       end_premise
    end_condition
    action sequential
       instigation sequential
         call this.mtProcess()
       end_instigation
    end_action
  end_rule
end_fbe
```

Code 3: Sensor FBE in NOPL

Fig. 6 shows the execution times for the sensor application, using ten thousand (10.000) sensor elements and varying the approval rates for the rules, meaning that only some of the sensors are activated in each iteration. An equivalent object-oriented implementation is also written to serve as a performance baseline for the tests. All tests were executed on a computer with a processor Ryzen 5 3600 at 3.6GHz and 16 GB of DDR4 RAM at 3000MHz.

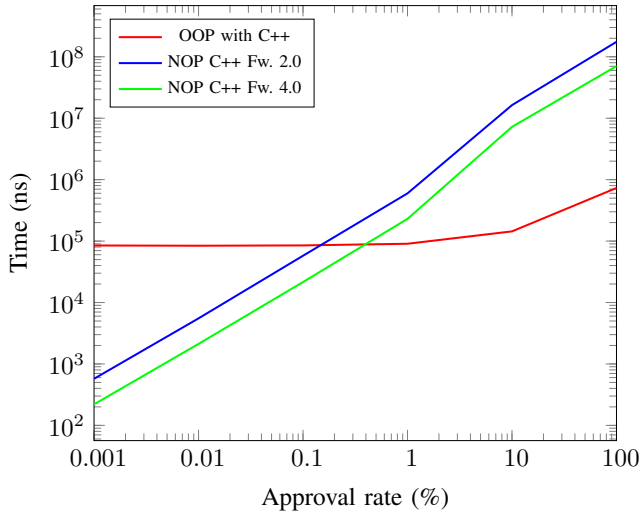

Fig. 6: Application benchmark

It is worth noticing how both NOP framework versions have similar performance behavior, with lower execution times than OOP when the approval rate is low, and higher execution times when the approval rate is higher. The OOP implementation, in its turn, has an almost constant execution time.

Besides, even if still somehow prototypal with applicability issues, current NOPL technology allow generating specific notifying code in C++ that achieves better execution time results than C++ under the usual loop-driven approach.

Another test was executed in order to evaluate memory usage for these applications. 10000 sensors were initialized, but not activated, multiple times, resulting in memory allocation and deallocation. The OOP application memory usage, as shown in Fig. 7, reached less than 2 MB during testing. Due to the memory capture process having a low resolution, and the memory consumption being low, it is not possible to clearly see the allocation and deallocation process.

The memory usage for the NOP C++ Framework 2.0 implementation is much higher, and in Fig. 8 it is also noticeable that there is a memory leak, since the memory consumption only increases across multiple test runs. A single test reached around 69MB of memory usage.

In comparison, the NOP C++ Framework 4.0 has a lower memory usage than the NOP C++ Framework 2.0, peaking at around 24MB, and properly deallocating memory across multiple runs, as it is possible to see in Fig. 9. Therefore, the NOP Framework C++ 4.0 reduces the memory consumption in over two thirds when compared to the NOP C++ Framework 2.0 for this application.

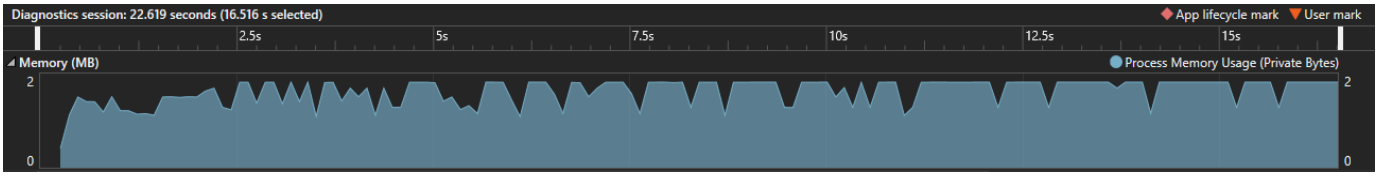

Fig. 7: OOP C++ application memory usage

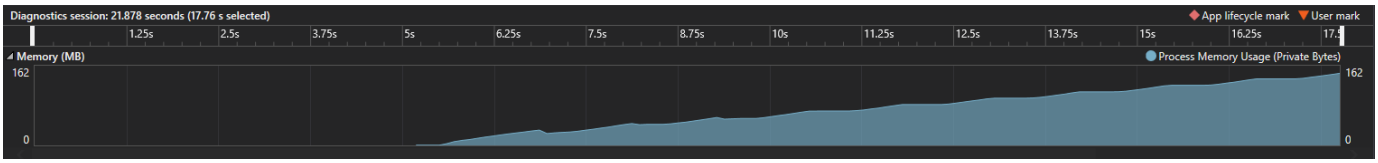

Fig. 8: NOP C++ Framework 2.0 application memory usage

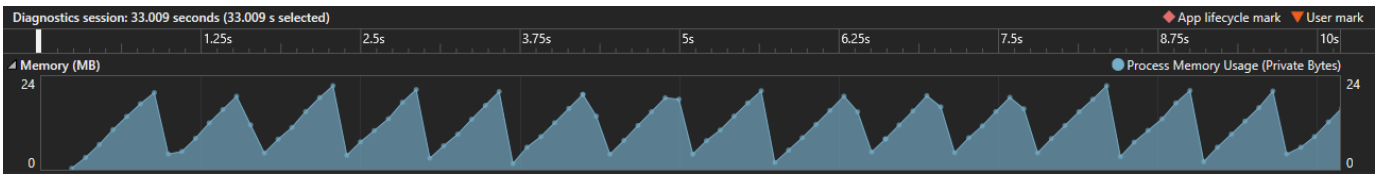

Fig. 9: NOP C++ Framework 4.0 application memory usage

VI. Conclusion

The NOP has multiple framework implementations, which are nowadays its stable implementations. However, the frameworks are not that easy to use, due to design limitations. By using modern C++ features it became possible to implement a new version of framework with a structure that is simpler and more generic, therefore easier to use. Moreover, it achieves superior performance when compared to the NOP C++ Framework 2.0 in the carried out experiment.

The NOP C++ Framework 4.0 reduces the amount of code required for the framework implementation by more than 92% when compared with the NOP C++ Framework 2.0. Also, when the execution times of the sensor application are compared, the NOP C++ Framework 4.0 shows a reduction of more than 50% in all the tests when compared to the NOP C++ Framework 2.0.

With the availability of a better framework, it is expected that more developers will choose to use the NOP to develop their applications, as a more flexible framework also allows for a wider range of applications to be developed. Besides, it is expected that the NOPL Technology will also incorporate the principles used to guide the elaboration of the NOP C++ Framework 4.0, achieving the proper synergy between them.

Acknowledgement

The authors would like to thank the research funding provided by the Araucaria Foundation (FA - Fundação Araucária), as well as the support from the Federal University of Technology (UTFPR - Universidade Tecnológica Federal do Paraná) in Brazil, particularly from the Informatics Department (DAINF - Departamento Acadêmico de Informática).

References

[1] P. Van Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*, 1st ed. The MIT Press, 2004.

[2] A. F. Ronszcka, G. Z. Valenca, R. R. Linhares, J. A. Fabro, P. C. Stadzisz, and J. M. Simao, "Notification-oriented paradigm framework 2.0: An implementation based on design patterns," *IEEE Latin America Transactions*, vol. 15, no. 11, pp. 2220–2231, 2017.

[3] R. R. Linhares, L. F. Pordeus, J. M. Simão, and P. C. Stadzisz, "Noca — a notification-oriented computer architecture: Prototype and simulator," *IEEE Access*, vol. 8, pp. 37 287–37 304, 2020.

[4] J. M. Simão, C. A. Tacla, and P. C. Stadzisz, "Inference process based on notifications: The kernel of a holonic inference meta-model applied to control issues," *IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans*, vol. 39, no. 1, pp. 238–250, 2009.

[5] J. Simão, "Notification oriented paradigm (nop) and imperative paradigm: A comparative study," *Journal of Software Engineering and Applications*, vol. 05, pp. 402–416, 01 2012.

[6] D. L. Belmonte, R. R. Linhares, P. C. Stadzisz, and J. M. Simao, "A new method for dynamic balancing of workload and scalability in multicore systems," *IEEE Latin America Transactions*, vol. 14, no. 7, pp. 3335–3344, 2016.

[7] G. Z. Valença, "Contribuition to the materialization of Notification Oriented Paradigm (NOP) via framework and wizard," Master's thesis, UTFPR, 2012.

[8] A. F. Ronszcka, "Contribution for the conceiving of applications in the Notification Oriented Paradigm (NOP) under the vias of patterns," Master's thesis, UTFPR, CPGEI, 2012.

[9] ——, "Method for the creation of programming languages and conpilers for the Notification-Oriented Paradigm in distinct platforms," Ph.D. dissertation, UTFPR, CPGEI, 2019.

[10] R. N. Oliveira, V. Roth, A. F. N. Henzen, J. M. Simao, P. Nohama, and E. C. G. Wille, "Notification oriented paradigm applied to ambient assisted living tool," *IEEE Latin America Transactions*, vol. 16, no. 2, pp. 647–653, 2018.

[11] F. Negrini, A. Ronszcka, R. Linhares, J. Fabro, P. Stadzisz, and J. M. Simão, "NOPL-Erlang: Programação multicore transparente em linguagem de alto nível," in *Anais da V Escola Regional de Alto Desempenho do Rio de Janeiro*, 2019, pp. 16–20.

[12] A. Ronszcka, C. Ferreira, P. Stadzisz, J. Fabro, and J. Simão, "Notification-oriented programming language and compiler," 2017, pp. 125–131.

[13] J. Dehnert and A. Stepanov, "Fundamentals of generic programming," vol. 1766, 01 1998, pp. 1–11.

[14] A. A. Stepanov and D. E. Rose, *From mathematics to generic programming*. Addison-Wesley, 2015.