ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 8, No. 1, 2023

PROPAGATOR-ORIENTED PROGRAMMING MODEL USING JAVA

Vladyslav Bilyk, Anatoliy Sachenko

Lviv Polytechnic National University, 12, S. Bandery str., Lviv, 79013, Ukraine
Authors' e-mails*: vladyslav.bilyk.mkisp.2022@lpnu.ua, as@wunu.edu.ua*

https://doi.org/10.23939/acps2023.01.009

Submitted on 11.03.2023

***Abstract*: The aim of this work is to explore and analyze an unconventional style of programming based on a propagator-oriented model of computation. The paradigm of propagation is characterized by networks of local, independent, stateless machines interconnected with stateful storage cells. This model allows for a highly modular design and multidirectional computation, enabling the creation of complex systems that can respond to changes and update their state accordingly.**

This work provides an overview of the propagator-oriented programming model, its motivations, and its advantages over other well-known alternative styles, using unsophisticated examples written in the Java programming language. We illustrate how propagator networks can be used to build flexible and efficient systems and present a basic framework for building such networks. The foundational components of the propagation model are implemented in Java as groundwork for the general-purpose framework.

We demonstrate the power of propagator-oriented programming through an example of a Pythagorean Theorem implementation. The example shows how the model can be used to build complex systems of an arbitrary number of constraints and cells. We highlight the importance of information propagation over limited linear computation and the benefits of the multidirectional computation enabled by propagator networks.

***Index Terms*: propagators; constraint programming; multidirectional computation; Java.**

I. INTRODUCTION

The field of computing has come a long way since the invention of the first programmable computer. However, the linear computing model that is at the heart of conventional computing is still limited in its ability to handle certain classes of problems. In particular, the imperative programming style that is typically used to write computer programs can be difficult to apply to problems that involve many interconnected variables. This is where the propagator-oriented programming model comes in.

Propagator-oriented programming is a computational paradigm that is designed to handle complex problems that can be presented as networks of independent nodes. The model is based on the idea of propagators, which are simple computational elements that can be used to represent the dependencies between variables. The key advantage of the propagator-oriented programming model is that it can handle many variables that are interdependent in a way of being both efficient and elegant.

Propagators lend themselves best to problems that can be presented as networks of independent nodes, but they are not limited to this kind of problem. The propagator-oriented programming model can be generalized to other kinds of problems, as long as they can be expressed in terms of dependencies between variables. This makes it a versatile tool for tackling a wide range of computational problems [1].

In this article, we will explore the propagator-oriented programming model in more detail. We will look at the limitations of conventional computing models and imperative programming styles and explain how propagators can be used to overcome these limitations. We will also examine an example of applying the propagator-oriented programming model and discuss the potential for future research in this area.

Over the past few decades, there has been significant research around constraint programming and logic programming, which are closely related to the propagator model of computation. These programming paradigms have been used to solve a wide range of problems, including scheduling, planning, and optimization.

- Constraint programming is a programming paradigm that involves specifying a set of constraints that must be satisfied by a set of variables. The goal is to find a solution that satisfies all the constraints. Constraint programming has the advantage of being able to solve problems that are difficult or impossible to solve with other programming paradigms. However, it can be computationally expensive and may struggle to handle problems with many variables or constraints [1]. One particular Java framework provides a platform for constraint programming in the form of a library [3]. It offers a declarative approach to problem-solving, allowing users to state the set of constraints that must be satisfied in every solution. The library uses a combination of constraint filtering algorithms and search mechanisms to efficiently solve the problem.
- Logic programming, on the other hand, involves specifying a set of logical rules that describe the relationships between different entities in a problem domain. The goal is to derive logical consequences of these rules. Logic programming has the advantage of being able to handle complex problems with many interdependent variables.

However, it can be difficult to express certain kinds of problems in a logical framework, and the search space can be very large [4].

- Reactive programming is another programming paradigm that is closely related to the propagator model of computation. Reactive programming involves programming with asynchronous data streams, where the program reacts to changes in the data stream. Reactive programming has the advantage of being able to handle real-time data streams and can be used to build responsive user interfaces. However, it can be difficult to reason about the behaviour of a reactive program, and it can be difficult to debug [5].

The propagator model of computation offers several advantages over these existing paradigms. Firstly, propagators are designed to handle many interdependent variables, making them ideal for problems that involve complex networks of dependencies. Secondly, the propagator model allows for the efficient propagation of constraints, which can lead to faster problem-solving times. Finally, the propagator model can be used to handle real-time data streams, making it a useful tool for building responsive user interfaces.

The propagator-oriented programming model has been the subject of intense research and development over the past few decades. One of the most notable contributions to this field [6] represents a significant milestone in the development of propagator networks.

The mentioned work provides a comprehensive and rigorous treatment of the propagator model, which is described as a general framework for reasoning about and comprehending complex software systems. The authors begin by introducing the basic concepts of propagator networks and demonstrating their flexibility and expressiveness through a series of examples. They then delve deeper into the theory of propagator networks, presenting a formal definition of the model and its associated algorithms.

One of its key contributions is the discussion of the relationship between propagator networks and other related programming paradigms, such as logic programming and constraint satisfaction. It is argued that propagator networks offer a more general and powerful framework for reasoning about complex systems than these other paradigms and provide several compelling examples to support this claim.

The research makes the first step into the untrodden territory that is the field of propagator-oriented programming. It is a testament to the power and elegance of the propagator model and has inspired a new generation of researchers to explore the possibilities of this programming style.

Although the work is undoubtedly fascinating and innovative, a major critique of the authors' approach is that the Scheme programming language, in which they designed and implemented the propagator-based model, is not widely used in the design of real-world systems [7]. This fact limits the applicability of their work to practical situations where performance and scalability are critical considerations.

Scheme is a dialect of Lisp, a language that is mainly used in academia and research settings [8]. While it has some advantages, such as being easy to learn and having a simple syntax, it is not a language that is widely used in industry. Most software development is done using languages such as Java, C++, Python, or JavaScript, which have more extensive libraries, are better supported, and are optimized for high performance and scalability [9]. Thus, while the propagator-based model framework implementation produced by Radul and Sussman may have theoretical value, it is unlikely to be adopted widely in real-world systems.

Hence the focus of this work is to implement the propagator-oriented model using Java, one of the most widely used languages in the industry [10]. Java is a popular language known for its robustness, portability, and scalability. Implementing the propagator-oriented model using Java would make it more accessible to a wider audience, including practitioners who are familiar with Java and are looking for innovative ways to solve complex problems and model complex systems [11].

II. PROBLEM STATEMENT

In the world of software development, the complexity of software systems is a well-known fact. With the increasing demands of modern software systems, the complexity of these systems is expected to increase. It is essential, therefore, that software systems are designed in a way that allows them to be flexible and evolvable. This means that software systems must be designed with the ability to adapt to new requirements, incorporate new technologies, and integrate with other systems.

Software systems are often developed in a rapidly changing environment, where the customer needs, and market demands can change rapidly. This requires software systems to be designed with the ability to evolve over time. A system that is designed to be flexible and evolvable will be able to respond to changes quickly and efficiently, while a rigid system will struggle to adapt and may quickly become obsolete.

Maintaining existing software systems can be a significant challenge in the world of software development. As these systems age, they become increasingly difficult to maintain, making flexibility a key requirement for long-term functionality. Without flexibility, a rigid system will require significant effort to keep it up-to-date and functional, while a flexible and evolvable system can be easily updated, enhanced, and maintained over time.

The problem statement that this paper addresses is how to design software systems that are flexible and evolvable. The propagator-oriented programming model offers a solution by representing dependencies between variables, allowing for the modeling of complex systems in an efficient and elegant way. This approach enables software systems to be updated and modified over time, without the need for significant effort to rewrite the entire system. This paper explores how propagator-oriented programming using Java can address the problem of maintaining flexible and evolvable software systems.

III. OBJECTIVES

The main objective of this work is to demonstrate the effectiveness of using the propagator-oriented programming model in Java to create more comprehensive and adaptable software models. Specifically, the aim is to reduce the time it takes to implement new features or modify existing ones in a software system. The following objectives are the central points of the study:

1. Explore the use of propagator-oriented programming model for building software systems and demonstrate its application using the Java programming language, highlight how propagator-based models can be effectively utilized to achieve higher efficiency and flexibility in software development.
2. Evaluate the effectiveness of the propagator-oriented programming model in terms of the time taken to implement new software features and modifications. This will be done by comparing the time taken to implement features using the traditional programming approach with that using the propagator-oriented programming model.

The study aims to reduce the average duration required for the incorporation of novel functionality and modification of existing components by a minimum of 20 %. Achieving these objectives will demonstrate the efficacy of the propagator-oriented programming model in creating more adaptable and comprehensive software models. Consequently, software developers will become familiar with a practical approach to software development, capable of adapting and evolving with changing requirements, while simultaneously reducing the effort and time required for the implementation of new features.

IV. PROPAGATOR-ORIENTED MODEL ARCHITECTURE

The propagator-oriented model is a computational model that is designed to handle problems that involve stateless autonomous cells. In this model, each cell is responsible for computing a value based on the values of its inputs and propagating its output to its dependent cells.

At a high level, the architecture of the propagator-oriented model can be divided into two main components: cells and propagators. Cells represent the stateless autonomous computation units that can accumulate information based on their inputs and their internal processing, and propagators are responsible for propagating the output of a cell to its dependent cells.

The propagator network is a key element of the architecture. It consists of a set of propagators that are responsible for propagating the outputs of cells through the network. Each propagator is connected to the cells that are involved in its computation. The propagator network can be thought of as a directed acyclic graph (DAG), where the nodes represent propagators, and the edges represent the dependencies between propagators.

Constraints are optional in this architecture. However, they are likely to be present, as practically every system is designed with some constraints in mind. Propagators are thus responsible for enforcing those constraints. When a cell's contents are modified, the associated propagators are alerted and possibly proceed to update the contents of dependent cells.

One of the distinctive features of the propagator-oriented model is the ability of cells to accumulate information based on their inputs and their internal processing state. Unlike variables that store a single result of a computation, cells can store and update intermediate results, which can be used to better understand and optimize the computation process.

In contrast to conventional linear models of computation, the propagator-oriented model is inherently nonlinear and can handle problems that involve complex networks of dependencies between cells. In this model, the problem is typically represented as a network of stateless autonomous cells, and the solution is found by propagating outputs through the network, as opposed to linear models of computation, where the problem is represented as a sequence of steps that are executed sequentially.

Consider a propagator network that models the expression of the Pythagorean Theorem: $a^2 + b^2 = c^2$.

Such a network can be represented as a directed acyclic graph with six cells: three for the values of a, b, c and another three for their squared values. Fig. 1 displays the DAG. Cell "a" and cell "b" represent the input variables, and cell c represents the output variable. Cell "a" and cell "b" are connected to the respective inputs of propagators modeling the square operation. The outputs of the intermediate results, a^2 and b^2, are connected to the propagator modeling the add operation, which takes two input cells and produces an output cell that represents their sum. Finally, the output of the add propagator is connected to the input of sqrt propagator that takes a square root of its input and stores the output in cell c, representing the hypotenuse length.

In this network, the intermediate results a^2 and b^2 are explicitly named and stored in dedicated cells. By storing the intermediate results, the network can be more easily understood and modified, as it separates the computation into smaller, more manageable units. Additionally, the intermediate result can be reused in other computations, which can lead to improved performance and reduced computational complexity. Although, in some cases, these memory requirements can be significant, and if the system's memory bandwidth is not sufficient, it could impact system performance. However, as the cost of memory continues to decrease and the availability of high-bandwidth memory increases, the impact of storing intermediate results on system performance is likely to become less significant over time.

In addition to the benefits for system design and performance, storing intermediate results in separate cells can also assist with the debugging process. In a propagator network, a change to one input cell can propagate through the network and affect multiple output cells. By storing intermediate results in separate cells, it becomes easier to identify which cells are affected by a particular change, as the intermediate results act as a checkpoint for the computation. This can help developers isolate and debug issues more efficiently, as they can focus on the cells that are affected by the change and ignore cells that are not directly related to the issue. Additionally, intermediate results can be inspected and monitored during runtime, providing valuable insight into the computation, and helping to identify any issues or anomalies in the system.

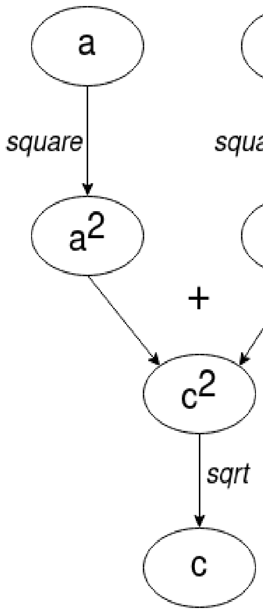

Fig. 1. Expression of the Pythagorean Theorem with propagators

Things get more interesting when we enable the bidirectional flow of information. At first glance, it may seem that the primary benefit of using cells to combine partial information is just a matter of cleaner aesthetics. After all, a problem such as the building example can be easily simulated in an expression language by adding an explicit computational step between the squaring of the inputs and the summation of the intermediate results. Although this approach produces a less incremental computation, as it must wait for all means of measurement to produce results before committing to the output, it can still work.

However, the real advantage of using cells lies in the ability to construct systems with a much broader range of potential information flows. By allowing cells to accumulate and combine information from multiple sources, we can enable multidirectional computation, where the results of one computation can feed into another, and vice versa. For example, the Pythagorean Theorem can benefit from multidirectional computation, as the computation of the length of the hypotenuse can feed into the computation of the length of one of the sides, and vice versa.

The Pythagorean Theorem network enhanced with bidirectional computation is presented in Fig. 2. It can be seen that primary input and output cells, as well as the cells storing the intermediate results, now propagate information in both directions. The network can now be used to compute the length of one of the sides once the cells for the hypothenuse and the other side know their values. The same can be said about the summation step, which now involves not only the add propagator but also the subtract propagator, inverting the information flow.

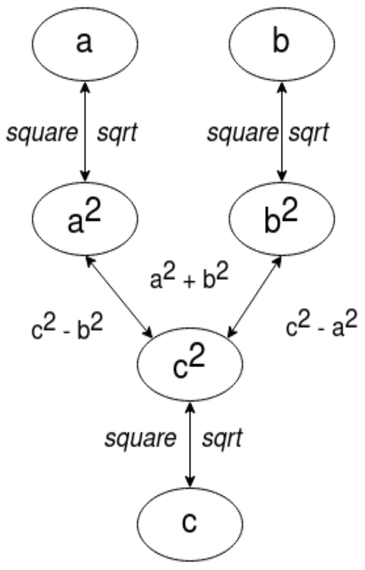

Fig. 2. Bidirectional information flow in the Pythagorean Theorem expression with propagators

V. SIMULATING THE EXAMPLE IN JAVA

Fig. 3 presents a Java class modeling the cell concept. It represents the foundational abstraction in the propagator-oriented programming model. A cell represents a node in the propagator network which accumulates information. The class has a generic type T that represents the type of information that the cell will store. The cell is also named and knows about propagators that need to be alerted whenever its value changes.

The value field stores the current value of the cell, and it is initialized as null, which has a special meaning in the propagator model. Null values indicate that a cell knows nothing about its value and thus should be treated carefully. The hasValue() method is designed for this purpose.

The setValue() method is used to change the contents of the cell. If the new value is different from the current value, then this new information must be propagated through the network. The method sets the new value and calls the propagate() method of each of the associated propagators, alerting the propagators that the value of one of its inputs has changed and that it needs to recompute its output.

The addPropagators() method is used to add propagators to the cell. The propagators list stores all the propagators that depend on the cell's value. Whenever the value of the cell changes, all connected propagators will be alerted.

```
class Cell<T> {
  public final String name;
  private T value;
  private List<Propagator> propagators;

  public Cell(String name) {
    this.name = name;
    this.propagators = new ArrayList<>();
  }

  public T getValue() { return this.value; }
  public boolean hasValue() { return this.value != null; }

  public void setValue(T value) {
    if (!value.equals(this.value)) {
      this.value = value;
      propagators.forEach(Propagator::propagate);
    }
  }

  public void addPropagators(Propagator... propagators) {
    this.propagators.addAll(Arrays.asList(propagators));
  }
}
```

Fig. 3. Java class representing a cell in the propagator network

The Propagator class in the propagator-oriented programming model, displayed in Fig. 4, is an abstract class that provides the foundation for all propagators in the system. This class serves as a blueprint for specific propagator implementations. The Propagator class contains two important fields – inputs and outputs – that represent respective cells connected to the propagator.

The constructor of the Propagator class takes two lists of Cell objects as arguments – the inputs and the outputs of the propagator. Additionally, the constructor adds the propagator instance being constructed to the list of propagators of each input Cell object. This ensures that whenever the contents of the input cells change, the propagator will be alerted.

The propagate() method of the Propagator class is an abstract method that must be implemented by each specific propagator. This method is called when any of the input Cell objects of the propagator change their value. When called, this method is expected to update the values of the output cells, ensuring that the propagator network remains consistent with the new input values.

```
abstract class Propagator {
  protected List<Cell> inputs;
  protected List<Cell> outputs;

  Propagator(List<Cell> inputs, List<Cell> outputs) {
    this.inputs = new ArrayList(inputs);
    this.outputs = new ArrayList (outputs);
    inputs.forEach(cell -> cell.addPropagators(this));
  }

  protected abstract void propagate();
}
```

Fig. 4. Java class representing a propagator

A specific propagator implementation is shown in Fig. 5. In particular, the addition operation is modeled. Every propagator implementation should declare a static register() method that configures the input and output cells by calling the internal constructor. There is no need for the programmer to interact with Propagator instances directly since they will be alerted whenever the contents of their associated cells change. The method propagate() is the core of any propagator, as it defines the actual computation steps. The Adder propagator, for example, first makes sure that both its inputs have values, and only then computes and sets the value of the output cell. Many propagators shall follow this style, as, typically, a computation makes sense only once all its inputs are provided with values.

```
class Adder extends Propagator {
  Cell<Long> in1; Cell<Long> in2;
  Cell<Long> out;

  public static void register(
      Cell<Long> in1, Cell<Long> in2, Cell<Long> out) {
    new Adder(List.of(in1, in2), List.of(out));
  }

  Adder(List<Cell<?>> inputs, List<Cell<?>> outputs) {
    super(inputs, outputs);
    this.in1 = (Cell<Long>)inputs.get(0);
    this.in2 = (Cell<Long>)inputs.get(1);
    this.out = (Cell<Long>)outputs.get(0);
  }

  protected void propagate() {
    if (in1.hasValue() && in2.hasValue())
      out.setValue(in1.getValue() + in2.getValue());
  }
}
```

Fig. 5. Propagator for the addition operation

To express multidirectional computation using atomic propagators, we need to decompose the computation into a series of smaller, directional computations. Each directional computation is represented by a specific propagator that connects input and output cells. These propagators are then combined in a network to form a larger, multidirectional computation.

To illustrate this approach, it is worth considering the summation operation involving two inputs and one output. This computation can be expressed as a form of multidirectional computation by decomposing it into three directional computations: the addition operation and two subtraction operations. The propagators required to express this are as follows:

1. Subtractor Propagator: This propagator takes two inputs and produces an output that is the result of subtracting the first input from the second.

2. Adder Propagator: This propagator takes two inputs and produces an output that is the result of adding the two inputs.

By expressing the summation operation in terms of 3 separate directions of information flow using propagators, summation can be modeled as a form of multidirectional computation. Fig. 6 illustrates the blueprint of this computation. The method sum() takes two cells representing inputs to be added and their result is stored in the third output cell. However, unlike simple addition, it also connects the output cell to both input cells using the Subtractor propagator. This way, whenever the output cell's contents change, the input cells are adjusted accordingly.

```
public static void sum(
    Cell<Long> in1, Cell<Long> in2, Cell<Long> out) {
  Adder.register(in1, in2, out);
  Subtractor.register(out, in1, in2);
  Subtractor.register(out, in2, in1);
}
```

Fig. 6. Bidirectional propagation via summation of two values

Fig. 7 illustrates the expression of the Pythagorean Theorem through a network of propagators. The Pythagorean class declares three fields of type Cell<Long> representing the input values *a*, *b*, and *c* of the theorem. It also declares another three fields of the same type representing the intermediate results a^2, b^2, and c^2, respectively.

```
class Pythagorean {
  Cell<Long> a; Cell<Long> b; Cell<Long> c;
  Cell<Long> a2 = new Cell("a^2");
  Cell<Long> b2 = new Cell("b^2");
  Cell<Long> c2 = new Cell("c^2");

  public static void register(
      Cell<Long> a, Cell<Long> b, Cell<Long> c) {
    new Pythagorean(List.of(a, b), List.of(c));
  }

  Pythagorean(List<Cell> inputs, List<Cell> outputs) {
    this.a = (Cell<Long>)inputs.get(0);
    this.b = (Cell<Long>)inputs.get(1);
    this.c = (Cell<Long>)outputs.get(0);
    quadratic(a, a2);
    quadratic(b, b2);
    sum(a2, b2, c2);
    quadratic(c, c2);
  }
}
```

Fig. 7. Expression of the Pythagorean Theorem using a network of propagators

The Pythagorean class has a static method named register that takes three cells, representing the input values of the Pythagorean Theorem, which instantiates the network by calling the constructor of the class. The constructor takes two lists of cells, representing the input and output cells of the theorem, respectively. It first extracts the input and output cells from the lists and assigns them to the corresponding instance variables. It then applies the quadratic propagator network to the input values "a" and "b", which similarly to the summation operation is a multidirectional operation. That is, it computes the power of 2 of its input cell's value and stores the result in the output cell. And it is also capable of performing the inverse operation, taking the square root of the output, and storing the result in the input cell. The summation propagator network is then applied to a^2 and b^2, which sets the value of c^2 to their sum. Finally, the quadratic network is applied to c, which sets its value to the square root of c^2.

Reflecting on the previously described Pythagorean class, two examples of its usage are presented. The first example, in Fig. 8, is quite straightforward; it expresses the computation of the hypothenuse length knowing the lengths of the other two sides of the triangle. The Pythagorean class instance is created, and values of "*a*" and "*b*" are then set to 3 and 4 respectively, which triggers the computation of *c* according to the Pythagorean Theorem. Since the value of *c* was previously unknown, it is computed and stored in its cell, and then outputted by the println statement. The answer produced by the network, as expected, is 5.

The second example, Fig. 9, is more interesting because it shows how the Pythagorean computation can be expressed in a multidirectional manner using a network of propagators. Here, *c* is set to 5 and *b* is set to 4, which should trigger the computation of *a*. The computation starts by calculating b^2, which is 16, and then subtracting that from c^2 which is 25. This gives us 9, which is stored in a^2. Finally, we take the square root of 9 to get *a*, which is 3. The fact that changing the value of the output cell *c* can trigger the computation of one of the inputs shows that the computation is in fact multidirectional.

The advantage of the propagator model is that it allows us to specify computations in terms of constraints rather than a linear sequence of steps. This makes it easier to specify complex computations involving many interdependent variables, and to update the computation efficiently when the values of the variables change. In the case of the Pythagorean Theorem, the propagator model allows us to specify the computation in terms of constraints on the values of *a*, *b*, and *c*, rather than in terms of a formula that computes *c* from "*a*" and "*b*". This makes it easier to reason about the computation, and to update the values of *a*, *b,* and *c* in any order, without having to explicitly compute the value of *c* each time the values of "*a*" and "*b*" change.

```
Cell<Long> a = new Cell("a");
Cell<Long> b = new Cell("b");
Cell<Long> c = new Cell("c");
Pythagorean.register(a, b, c);
a.setValue(3L);
b.setValue(4L);
System.out.println("c = " + c.getValue());
```

Fig. 8. Calculating the hypotenuse length

```
Cell<Long> a = new Cell("a");
Cell<Long> b = new Cell("b");
Cell<Long> c = new Cell("c");
Pythagorean.register(a, b, c);
c.setValue(5L);
b.setValue(4L);
System.out.println("a = " + a.getValue());
```

Fig. 9. Calculating the length of one of the sides

The Pythagorean Theorem example is just a simple illustration of how propagator networks can be used to express and solve a system of constraints. In more complex systems, there may be many interdependent variables, each with its own set of constraints. As the number of variables and constraints grow, traditional approaches to solving such systems become increasingly cumbersome, error-prone, and time-consuming.

In propagator-oriented programming, the goal is to express the constraints as propagators, which can then be composed into a network. Each propagator represents a single constraint or a computation, and the propagators interact with one another to collectively enforce the system's constraints. By propagating information in a multidirectional way through the network, propagators can update their inputs and outputs in response to changes in other parts of the network. This allows for complex, interdependent systems of constraints to be modeled and solved in a flexible, modular way.

The key advantage of this approach is that it emphasizes information propagation rather than limited linear computation. Instead of explicitly solving for each variable in a step-by-step fashion, propagator networks use constraints to implicitly enforce the relationships between variables, allowing them to update and propagate information in a more flexible and adaptive manner. This allows propagator networks to efficiently handle complex systems with many variables and constraints, and to provide a more elegant, maintainable, and scalable solution than traditional approaches.

VI. EVALUATION

To compare the effectiveness of the propagator-oriented programming model with a conventional object-oriented approach in a real-world scenario, an experiment was conducted on the development of a software tool that performs calculations, organizes and manipulates data, and presents information in a tabular format, with an interface resembling that of a spreadsheet. The experiment involved two teams of software engineers from the industry, with approximately equal levels of skill and experience in Java programming. All members had a minimum of five years of experience in software development, with expertise in software design, development, and testing. One team employed a conventional object-oriented approach to software development, while the other team was introduced to the propagator-oriented model and had a week prior to the beginning of the experiment to get familiar with the model. The experimental development process comprised an incorporation of five distinct user stories that encompassed various functional requirements. Three of the user stories mandated the incorporation of new functionality, while the remaining two required the modification of significant segments of pre-existing code.

The team that used the conventional object-oriented approach implemented the fundamental version of the application within 9 days. Incorporating each novel feature mandated an additional timeframe ranging from two to six days for successful implementation. However, incorporating breaking changes led to a marked decrease in the development pace, requiring complete attention of all team members, and leading to a supplementary 13 days of development time. In summary, the entire development process lasted for a total of 24 days.

On the other hand, the team that adopted the propagator-oriented model developed a basic version of the application within a period of 10 days. Incorporating new features demanded a timeframe ranging from one to four days for successful implementation. Moreover, the incorporation of breaking changes did not pose significant challenges, as the system exhibited notable flexibility and adapted seamlessly to such modifications. Consequently, the entire development process was completed within a timeframe of 20 days.

The comparative evaluation of both approaches adopted for the development of the application indicates a noteworthy reduction of 17 % in total development time, and 30 % in duration required for implementation of new features and modification of existing ones to meet user requirements.

As is characteristic of any software system, new feature requests from users are likely to occur as the system evolves over time. Based on the experiment conducted, it is evident that the propagator-oriented model represents a more efficient approach towards maintenance and evolution of software. Therefore, it is reasonable to expect that the difference in development time between both approaches will increase as the system grows in complexity, and more changes are necessary. The propagator-oriented programming model will thus become progressively more valuable as the system expands and develops, offering a more inclusive and adaptable framework for software development.

VII. CONCLUSION

In conclusion, the propagator-oriented programming model presents a nontraditional approach to solving complex problems by building systems that can handle an arbitrary number of constraints and cells, and support the flow of information in more than one way. Unlike the conventional linear computation style, the model emphasizes the importance of information propagation and the interdependence of variables, allowing for more flexible and comprehensible systems.

Through the Pythagorean Theorem example and its generalization, it can be observed how this model can be applied to various real-world problems, ranging from building systems that solve engineering problems to designing systems that lend themselves better to informal reasoning. The model offers a framework for building reactive, logic, and constraint programming systems that can adapt and update in response to changes.

The outcomes of the experiment demonstrate a 30 % decrease in the duration required to implement novel functionality and modify existing features to meet user requirements.

REFERENCES

[1] A. Kulkarni, M. Lang, A. Lumsdaine (2011). GoDEL: A Multidirectional Dataflow Execution Model for Large-Scale Computing. *First Workshop on Data-Flow Execution Models for Extreme Scale Computing*. Pp. 10–18 [Online]. DOI: https://doi.org/10.1109/DFM.2011.12

[2] Belaid, M.-B., Bessiere, C., Lazaar, N. (2019). Constraint Programming for Association Rules. *Proceedings of the 2019 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics* [Online]. Pp. 112–125. DOI: https://doi.org/10.1137/1.978161 1975673.15

[3] Prud'homme C., Fages J. (2022). Choco-solver: A Java library for constraint programming. *Journal of Open Source Software*. Pp. 1–5 [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.04708.pdf (Accessed 02/21/2023)

[4] Baumgartner, P. (2021). The Fusemate Logic Programming System (System Description). *ArXiv*. Pp. 85–107 [Online]. DOI: https://doi.org/10.48550/ ARXIV. 2103.01395

[5] Perez, I., Goodloe, A. E. (2020). Fault-tolerant functional reactive programming (extended version), *Journal of Functional Programming. Cambridge University Press (CUP)*. Pp. 57–72 [Online]. DOI: https://doi.org/10.1017/ s0956796820000118

[6] Radul A., Sussman G. J. (2009). The art of the propagator. *Proceedings of the 2009 international lisp conference*. Pp. 15–31 [Online]. DOI: https://dspace.mit.edu/ handle/1721.1/44215

[7] Radul A. (2009). Propagation Networks: A Flexible and Expressive Substrate for Computation. *Computer Science and Artificial Intelligence Laboratory Technical Report. Massachusetts Institute of Technology*. Pp. 75–98 [Online]. DOI: https://dspace.mit.edu/handle/1721.1/49525

[8] Lilis, Y., Savidis, A., (2019). A Survey of Metaprogramming Languages, *ACM Computing Surveys*, 52(6). Pp. 1–39 [Online]. DOI: https://doi.org/ 10.1145/3354584

[9] Bissyandé, T.F. *et al.,* (2013). Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects, *Computer Software and Applications Conference*. Pp. 303–312 [Online]. DOI: https://doi.org/ 10.1109/compsac.2013.55

[10] Taboada, G.L. *et al.* (2013). Java in the High Performance Computing arena: Research, practice and experience. *Science of Computer Programming*, 78(5). Pp. 425–444. [Online]. DOI: https://doi.org/10.1016/j.scico.2011.06.002

[11] Marii, B., Zholubak, I. (2022). Features of Development and Analysis of REST Systems. *Advances in Cyber-Physical Systems,* Vol. 7, No. 2. Pp. 121–129. DOI: https://doi.org/10.23939/acps2022.02.121

Vladyslav Bilyk earned his Bachelor's Degree in Systems Analysis at the Ukrainian Catholic University. Currently he is earning a Master's Degree in Systems Programming at Lviv Polytechnic National University. He is interested in the programming language theory, software engineering and functional programming.

From 2022 and to the present he has been a software engineer at Fielden Management Services.

Anatoliy Sachenko is a Professor of Department for Information Computer Systems and Control, West Ukrainian National University and Professor of Department for Informatics and Teleinformatics, Kazimierz Pulaski University of Technology and Humanities in Radom, Poland. He earned his PhD Degree in Electrical Engineering at Lviv Physics and Mechanics Institute, Ukrainian National Academy of Science, 1978 and his Doctor of Technical Sciences Degree in Electrical and Computer Engineering at Leningrad Electrotechnic Institute, 1988. Research areas: Computational Intelligence, Distributed Measuring Systems, Intelligent Cyber Security, Wireless Sensor Networks, IT Project Management.