Object-Process Programming
A Visual Programming Language for Complex Systems Design and Implementation

In Partial Fulfilment of the Requirements for the Degree of

Doctor of Philosophy

Arieh Bibliowicz

Submitted to the Senate of the Technion – Israel Institute of Technology

Haifa, Tevet, 5777, January, 2017

This research thesis was done under the supervision of Professor Dov Dori in the Faculty of Industrial Engineering and Management

The generous financial support of the following organizations is gratefully acknowledged:

Technion – Israel Institute of Technology

European Union FP7 – VISIONAIR

Table of Contents

List of Figures

ABSTRACT

Software systems are a part of our daily life. As time goes by, these systems have become more complex, and this complexity has made their development ever more difficult. Several methods have been proposed to reduce this complexity, one of them being the use of visual modeling tools that provide the developer with ways to better understand the system and share how it works for other stakeholders within and outside of her or his team. In this work, we present OPP – Object-Process Programming, a full-fledged visual programming language based on the graphical language used by Object-Process Methodology. OPM has been shown to have a robust basis for modeling complex systems, and is being used in different and diverse industries, therefore its use as a programing language is a natural extension for it.

We performed two experiments to evaluate OPP. The first experiment was done with a group of 104 undergraduate students and the second as a focus group discussion with six professional developers and system engineers who are familiar with OPM and various programming languages. In the first experiment, the students programed a simple system using OPP. The students found value in the simplicity, readability, and visual representations used in the language. Most students thought that the language could not be used for real world programming as it requires a lot of work to implement simple programs. In the second experiment, a case study was presented to the focus group, and after this a discussion sessions was performed, focused on the positive and negative aspects of the language. The focus group found the value in OPP in the systems engineering area, to create a holistic view of the system which is also executable. Similarly to the student cohort, the experts thought that using OPP for low-level programming is not valuable.

Based on the qualitative and quantitative feedback given by the students and the experts, we are currently developing a new version of the language with improved semantics, a more user-friendly interface, and more built-in processes in the language runtime.

1 INTRODUCTION

Visual programming languages (VPLs) have long captured the imagination of software engineers and researchers as a tool to help simplify software development. Since the early 1960s [1], many different programming approaches have been taken to create new VPLs, but as of to-day, most programming is still done using textual programming languages, and in some cases even in the same languages that were developed more than 40 years ago.

Some 20 years ago, software systems modeling started being proposed as one of the solutions to help manage the increasing complexity of software systems, which have become part of our daily lives. The idea underlying software systems modeling is that it can raise the level of abstraction of the system from the implementation details to the domain details, thus focusing on the domain of the problem for which the system is being developed. This idea has been termed Model-Driven Architecture [2] [3] by the Object Management Group [4], Model-Driven Engineering [5], Model-Driven Software Development [6] and other names.

The current state of the practice is that models are used primarily for upfront design, after which the models cease to be maintained [7]. This is a very problematic situation, since most of the cost for software is caused by software maintenance [8], and having outdated models, which can be misleading or plainly wrong, may even raise the cost of maintenance instead of reducing it. As stated by Gueheneuc et al. [9], "A recurrent problem in the object-oriented software engineering community is the transition between a software design and its implementation, and vice versa, during the implementation and the maintenance phases."

One solution to break this model-program disconnect is mixing these two fields, creating a visual programming language that can be used for system modeling, software development and execution.

This work describes the principles, definition, applications and evaluation of Object-Process Programming (OPP) language, a fully executable VPL based on the graphical and textual language used by Object-Process Methodology (OPM) [10], which as of 2015 is ISO 19450 [11]. We have chosen to use OPM as the basis for our language because it has been shown to be able to model real-world problems from different domains, such as biological systems [12], ERP [13], Web Applications [14], and Domain Specific Modeling Languages (DSMLs) [15].

2 BACKGROUND AND RELATED WORK

This section presents research that has been performed in fields related to this work: visual programming languages, model-driven software engineering, and Object-Process Methodology.

2.1 VISUAL PROGRAMMING LANGUAGES

There are multiple ways to define what visual programming is. Burnett [17] defines that "Visual programming is programming in which more than one dimension is used to convey semantics." A similar definition is given by Myers [18], who states that "visual programming refers to any system that allows the user to specify a program in a two (or more) dimensional fashion." A different definition is given in Wikipedia [19], which states that "a visual programming language (VPL) is any programming language that lets users create programs by manipulating program elements graphically rather than by specifying them textually."

Many visual programming languages have been developed throughout the years, yet most of them have not become mainstream for general purpose programming. The Wikipedia category page for VPLs [19] lists 75 different languages as of Sept. 2015. Eric Hosik [20] maintains a comprehensive list of VPLs that contains more than 158 languages as of Feb. 2014, starting with Sketchpad [1] , which is considered by most to be the first VPL, the list goes on to include more well-known diagramming languages, such as Flowcharts [21], and Statecharts [22], and culminates in modern languages and environments, such as Alice [23], Scratch [24], and LabView [25].

Some researchers have found that visual programming languages are easier to understand than textual languages. A controlled experiment done by Scanlan [26] showed that structured flowcharts greatly outperformed pseudo-code in program understanding, especially as complexity increased. A very similar result was found in an analysis performed by K. N. Whitely [27], which concluded that visual languages are better than textual languages in problem-solving situations, especially when the size or complexity of the problem grows.

Despite these findings, visual programming languages have not yet revolutionized software programming. In the classical paper "No Silver Bullet", Brooks said that "favorite subject for PhD dissertations in software engineering is graphical, or visual, programming… Nothing even convincing, much less exciting, has yet emerged from such efforts" [28]. Two main reasons he gives for this are that (1) computer screens are too small to show a seriously detailed software diagram, a problem that may have been solved with current high resolution 24+ inch flat screen displays, and (2) because software is difficult to visualize. But even while computer screens are now much larger than those from two decades ago, visual programming languages still require more space than textual languages to display the same

amount of information, as was analyzed in [29]. In that research, visual and textual languages were compared using token density, and the conclusions were that visual languages are not viable for general purpose programming, since to achieve the same token density as textual languages the VPL diagrams would need to become obscured and too complicated to understand, as measured by a "confusion count" rating. This can be seen as a confirmation of the so called "Deutsch Limit", which states that "The problem with visual programming is that you can't have more than 50 visual primitives on the screen at the same time" [30].

Some research efforts that have focused less on complete visual languages and more on their parallel uses in software development suggest that visual languages help the developers create better mental representations of their programs [31], and also to better understand code [32]. A study by Petre [33] showed that visual representations added value for experienced programmers, but only when the visualization was tailored for the user's task and goal.

The research on visual programming languages is ongoing, and as development platforms and hardware (including displays and input devices) improve, we expect to see new and novel developments in this field. This work is a step in this direction.

2.2 Model-Driven Engineering

Modeling for software engineering was a topic of great interest in the 1980's with the use of computer aided software engineering (CASE) tools, which were promoted as "one of the solutions that will counter the problems of poor software quality" [34]. These tools promised a lot, yet to this day most of the development of software systems is still done directly by programmers in 3rd or 4th generation programming languages.

There is renewed interest in this field, aimed at taming the complexity of the system and reducing some of the problems from which software system development suffers. These methodologies are called Model-Driven-X, or MDX: Model Driven Development (MDD), Model Driven Design (MDD), Model Driven Engineering (MDE) and Model Driven Architecture (MDA), the latter being a trademark of the Object Management Group (OMG) – creators of UML – the Unified Modeling Language [35]. These approaches are based on the assumption that the software industry cannot continue creating ever more complex software systems without models to abstract the systems being developed.

The most accepted definition for MDX is MDA – Model Driven Architecture [3] [36], proposed by OMG [4], a non-profit consortium that sets standards for distributed object-oriented systems, which currently focuses on creation of standards for modeling of systems. The MDA proposes a forward-engineering methodology, where systems are built using models and model transformations as follows:

1. A system is first described in a Computational Independent Model (CIM), which does not show details of the structure of the system, but describes its working

environment and its interactions with it. This model, sometimes called the Domain Model, is usually created using a vocabulary that is natural to the practitioners of the domain to which the model adheres. The CIM is used to bridge the gap between the users of the system – those that provide the requirements, and the development architects of the system – who translate these requirements into system architecture. A CIM can consist of many models. CIM requirements should be traceable from and to the PIM and the Platform Specific Model (PSM), described below.

2. After the CIM is created, a Platform Independent Model (PIM) is created, based on the requirements defined in the CIM. A PIM describes how a system is built without specifying the platform-specific details of its implementation.
3. The last model in the MDA chain is the Platform Specific Model (PSM), on which all the details related to a specific platform are added. A platform can be a specific technology (Java, .NET), a programming framework (J2EE, COM), a hardware platform (UNIX, Windows), or some mix of them.

The idea behind MDA is that the PSM should be directly derived from the PIM using model mappings. A mapping provides a specification for transforming a PIM into a PSM for a particular platform. Elements in the model can be marked for specific interpretation by the transformation. MDA provides the following added values:

1. When new technologies become available, they must be added only to the model mapping, so all the existing PIMs can start using this new technology.
2. Integration of existing technologies is done at the mapping level, whereas the modeling of the system is independent of them, relieving the developers of this complexity.
3. The maintenance of the system becomes easier thanks to the availability of a machine-readable design.
4. Some of the testing and simulation tasks can be done faster if there is already a platform-specific mapping for these purposes.

The OMG proposes UML as the MDA modeling. While MDA proposes a novel and simple methodology for system development, one of its basic problems is that it is based on UML, a complicated modeling language. Indeed, the full infrastructure [35] and superstructure [35] definitions of UML add up to over 900 pages of technical reading. This size and complexity has been shown to have quite a few negative aspects:

1. “UML 2.0 lacks both a reference implementation and a human-readable semantic account to provide an operational semantics” [37].
2. “UML diagrams are beset with duplications, which not only threaten the clarity and explicitness of the object definition but also waste valuable time and human resources” [38].

3. In a survey supported by the OMG, done in 2003-2004, , it was found that basic UML behavioral diagrams, such as Sequence diagrams and Use Case diagrams are not used because they are “not well understood by analysts” and had “insufficient value to justify the cost” [39]. Without a good definition of system behavior, the amount of value that can be derived from a model is fairly reduced.
4. Navigating the large UML 2.0 meta-model is a “time-consuming and frustrating task” [40].
5. “The numerous modeling concepts, poorly defined semantics, and lightweight extension mechanisms that UML provides make learning and applying it in an MDD environment difficult” [41].
6. “A major problem with UML is the size of its alphabet… it lacks system-theoretical ontological foundation… Lack of UML support for integrating structure and behavior in a single model puts the intellectually demanding burden of flipping back and forth between at least two diagram types entirely on the developer’s shoulders” [42].

Another problematic aspect of MDA is that the model and the system are connected by forward transformations, after which the parts of the system that were not defined in the model are implemented manually by the software developers. As the implementation process continues, the model of the system becomes outdated, either because its maintenance cost is considered to be too high or because there is no methodology supporting roundtrip MDA [43]. When such divergence starts, the usefulness of the model decreases sharply, especially as a basis for creating future releases of the software system. As stated by Gueheneuc et al. [9], “A recurrent problem in the object-oriented software engineering community is the transition between a software design and its implementation, and vice versa, during the implementation and the maintenance phases.”

While UML is the most well-known modeling notation in the software community in general, there are other modeling languages, tools and methodologies for Model-Driven Development, including the following:

1. The AmmA modeling toolbox proposal [44], an evolution of the AMMA platform [45], is a modeling platform designed for both forward and reverse engineering for MDD. The toolbox consists a low-level interpreted modeling framework (KMF – kernel modeling framework), a common modeling runtime (CMR) that executes model transformations, projectors used to load and store external models into the toolbox so that different modeling languages can be used in the model transformation process, and a global management unit to control the high-level model handling. Model transformation is done using the AtlanMod Transformation Language (ATL).

2. StateML+ [46] uses a modeling language based on state machines to generate executable, parallel and thread safe ADA code.
3. Fernandes et. al. [47] integrated Data Flow Diagrams (DFDs) with UML using an MDD approach, where the DFD is automatically mapped to object diagrams that can be used in the model transformation chain.

There are yet other modeling languages, but since it is hard to find evidence of their use outside the research community, it appears that most MDD development in industry, to the extent that it exists at all, is done using UML.

2.3 Object-Process Methodology

Object-Process Methodology (OPM) [10] is a conceptual modeling approach for complex systems that integrates in a single view the functional, structural and procedural aspects of the modeled system using formal yet intuitive graphics that is translated on the fly to a subset of natural language.

The basic principle behind OPM is that both objects and processes are first class entities which are needed to model a system. To model the behavior of the system, OPM uses procedural links that connect objects and processes to describe flow of control, information or material. Together, the objects, processes and links coexist in a single model that integrates both the structural and procedural aspects of the system, a trait which "reinforces the user's ability to construct, grasp, and comprehend the system as a whole and at any level of detail" [48]. To further help system comprehension, OPM provides three refinement-abstraction methods to cope with model complexity: in-zooming and out-zooming provides for the refinement of complex entities by hiding its components at high abstraction modeling levels and showing them when their details are required; unfolding and folding provides a means to model the structure of system entities separate from their behavioral aspects while still keeping them in the same model; and state expressing and suppressing gives freedom to show or hide the states of an object as desired. These mechanisms enable the OPM modeler to specify and refine the system indefinitely to any desired level of detail without losing legibility and maintaining simplicity at every detail level.

A novel feature of OPM is its bimodal representation which consists of both a graphical and a textual representation of the model that is automatically created based on the graphical representation. This bimodal representation allows for dual-channel processing of the model thus improving model comprehension [49], and tailors to both technical oriented stakeholders that prefer models and non-technical stakeholders which are more comfortable using textual descriptions.

Because of its intrinsic integration of structure and behavior, OPM provides a solid basis for modeling complex systems, and has been extended to model real-time systems [50], ERP

[13], multi-agent systems [51], data warehouses [52], biological systems [12], and Web applications [14].

OPM is defined by its reflective metamodel [48]. A metamodel is a model of modeling language which provides further understanding of the modeling language and provides a robust basis for code generation, model transformation and analysis. Furthermore, OPM's metamodel is fairly compact, and easy to extend and customize. The modeling syntax of OPM has also been formalized using graph-grammars and syntax checking algorithms [53].

OPM modeling is supported by the OPCAT [54], a collaborative modeling CASE environment.

3 THE OBJECT-PROCESS PROGRAMMING LANGUAGE

3.1 INTRODUCTORY EXAMPLE

We begin the description of OPP with a simple “Hello World” example program. An OPP program consists of a set of Object-Process Diagrams (OPDs), which are the canvases where the program is defined. Our first program consists of only one OPD, shown in Figure 1.

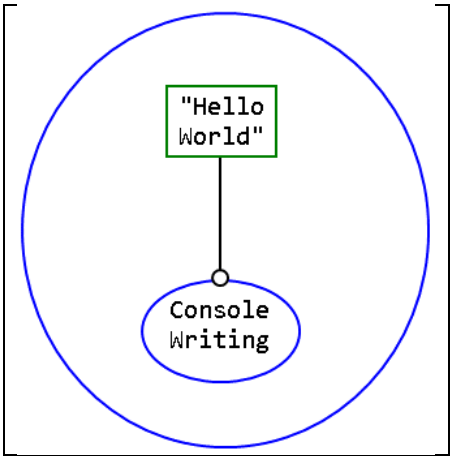


Figure 1 – The first OPP program: “Hello World”

An ellipse (which is blue by default) defines a process, which is the executable entity of the language. The big ellipse is an In-Zoomed process that defines an executable unit of the program. The rectangle (which is green by default) is an object, which stores data. This object is initialized with the value "Hello World". The object is connected with an instrument link to a process called “Console Writing”, so the object’s value is passed to the process as an argument when the process is executed. As there is only one process inside the In-Zoomed process, it is executed as soon as the In-Zoomed process is executed, and the result of running this program is the text “Hello World” written to the console of the computer.

3.2 LANGUAGE ELEMENTS: OPDS, NODES, AND LINKS

This section describes the graphical building blocks of an OPP program. While the OPP language is based on OPM, the full richness of the OPM syntax has not been implemented in OPP as the definition of OPP was done in parallel to the definition of OPM ISO/PAS 1945:2105, and there are some sematic questions that need to be resolved. Future work is being done to align the OPP language with OPM as it is currently defined in ISO/PAS 1945:2015 [11].

3.2.1 The Diagramming Canvas – Object-Process Diagram

An Object-Process Diagram, abbreviated OPD, is the canvas where OPP programs are defined. Its visual representation is a blank two-dimensional area. OPDs have no graphical representation. They are the canvas within which the other OPP elements are depicted to create programs. An OPD can contain two kinds of elements: Nodes and Links.

There are four kinds of OPDs: System OPD, Type OPD, Interface OPD, and In-Zoomed OPD.

1. A System OPD is the entry point of any OPP program. It defines which In-Zoomed processes are exposed by the program, which are internal to the program, and which are executed without external intervention when the interpreter loads the program.
2. A Type OPD defines non-basic OPP types, which are possibly complex data structures in an OPP program. OPP has two basic types: number and string. Using these, along with objects, processes, and structural links, the programmer can define new types in the Type OPD and then use these types in the system being developed. OPP is optionally-typed, meaning that the programmer is not required to define the specific types of data that are used in a program. However, defining the types used by the program allows the development environment and interpreter to find data-type inconsistencies and warn or even remove type-related errors.
3. An Interface OPD determines and shows the external interface of a process, which is a set of formal parameters of a process (which can be an In-Zoomed process or an OPP built-in process), its possible inheritance hierarchy, and pre- and post-executing processes.
4. An In-Zoomed OPD is the executable unit of the language. Within an In-Zoomed OPD the programmer defines the set of lower-level processes (sub-processes) that are performed when the process is executed by the interpreter, how the parameters are used, and what internal data is defined.

A complete definition of these OPD kinds appears in Sections 3.3, 3.5, 3.6, and 3.7, after the basic elements of the language are described.

3.2.2 Nodes - Objects, Processes, and States

Objects and processes are the nodes in an OPD. States are second-level nodes that can only exist inside Objects. The visual representation of these nodes is shown in Figure 2.

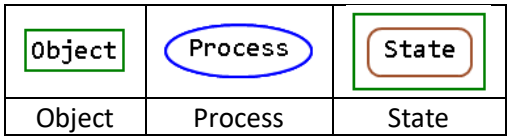


Figure 2 – OPP nodes: Object, Process, and State

3.2.2.1 Objects

Objects are used to define types and where data is stored in a program, depending on the type of the OPD where they are used. An object is represented by a green-bordered rectangle.

An object can have text inside it of the form *[name][:type][[=]value]*, where elements inside the brackets ('[' and ']') are optional. The first part of the text is the name of the object, which can be any valid string in any language (including spaces), excluding the ':' character, which is used to separate the name from the type, and '=', which is used to assign a value to an object. The second part of this text is the type of the object, which can be one of the OPP language basic types (Number or String) or a type that was defined in a Type OPD. The object name is optional, because its identity is defined by its location in the OPD. For this same reason, two objects in an OPD can have the same name, yet be considered different objects by the interpreter. An object that has neither a name nor a type is called an *anonymous object*. Examples of the naming possibilities are shown in Figure 3.

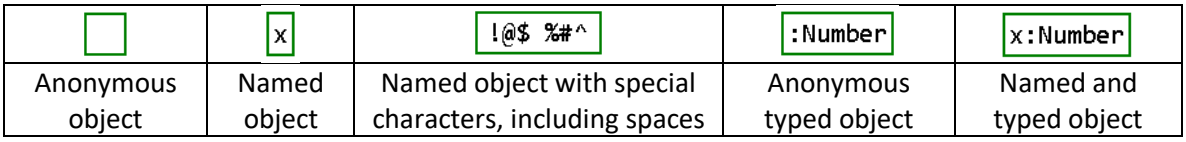


Figure 3 – Object naming examples

The third part of the object text is a value with which the object is initialized. OPP supports multiple ways to initialize an object at definition time. Two examples of these are numbers and strings, as shown in Figure 4. Section 4.1 expands on the use of initial values for objects and the syntax supported.

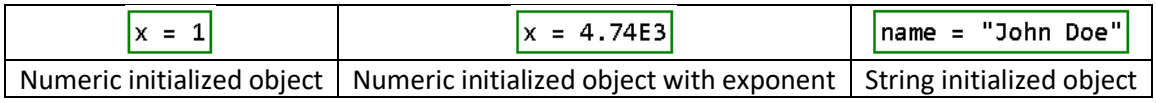


Figure 4 – Numeric- and string- initialized objects

When used as data stores, objects can have local or global scope. A locally-scoped object has a value that is accessible only in the OPD where it is defined. Conversely, a globally-scoped object has a value that can be accessed from any OPD in the system. A global object must have a name. Global objects are represented visually by adding a shadow to the object, as shown in Figure 5.

Figure 5 – Globally-scoped object

An object can be defined as a constant, in which case the value of the object cannot be changed during the execution of the program. An object can also be defined to contain a value, without having to name it or specify its type, and in this case it is automatically a constants. A constant object is represented visually by a changing the border of the object to a dashed border, as shown in Figure 6.

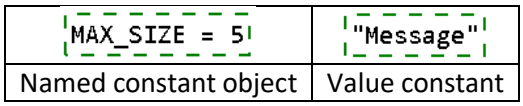


Figure 6 – Constant object examples

An object can be both global and constant, in which case it is represented with a dashed border and a shadow, as shown in Figure 7.

Figure 7 – A global constant object example

3.2.2.2 Processes

Processes are the executable units of the language. A process is represented by a blue-bordered ellipse. A process can have a name, which can be any valid string in any language (including spaces). When a process is In-Zoomed, the name of the process is the same as the name of the OPD where it is defined, and in this case the name of the process is not shown inside the process ellipse. Examples of the naming possibilities are shown in Figure 8.

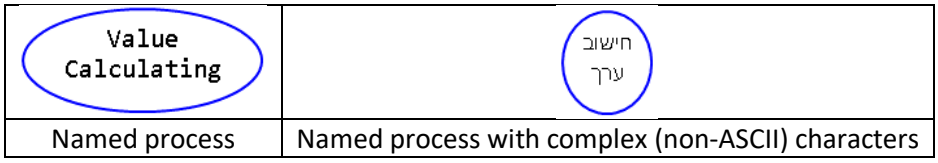


Figure 8 – Process naming examples

A process can be either abstract or concrete. A concrete process is a set of instructions that the interpreter can execute. An abstract process can define properties and parameters, which can later be implemented by one or more concrete processes. An abstract process can also be used to jointly represent concrete processes that conceptually have a common functionality. While a concrete process has a solid contour, an abstract process is represented by a dashed contour, as shown in Figure 9.

Figure 9 – An abstract process

3.2.2.3 States

States represent possible situations that the object can be during its lifetime or values than an object can attain. States are used in conjunction with procedural links to control the flow of execution of an In-Zoomed OPD. A state is represented by a brown-bordered rounded rectangle. States are only valid inside an object that does not contain any parts and which is of a simple type (Number or String), as defined in Section 3.5.

A state must contain text. If the object containing the state is typed, the text in the state must match the type of the object. An object is in a specific state if the current value stored in the object matches the value of that state.

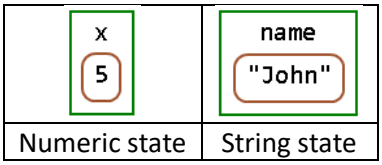


Figure 10 – Numeric and string state examples

In Figure 10, the object **x** will be at state **5** when the value of the object is set to 5. Likewise, the object **name** will be in state **"John"** if the value of the object is set to the string "John".

States can have logical conditions defined by numerical and string comparisons. A state can have numerical comparison operators, for example $x < 5$. When using these operators, the letter x refers to the current value of the object. The following numeric comparison operators are supported in OPP: $x < n$, $x <= n$, $x >= n$, $x > n$, and $x\ != n$, where n is any valid number. The example in Figure 11 shows an anonymous object with the state $x < 5$, therefore the object will be in this state when its value is a number that is less than 5.

Figure 11 – A numerical condition state example

A state can also have a string comparison operator using regular expressions. A string comparison is defined by preceding the string of the regular expression with the ~ character, as shown in Figure 12.

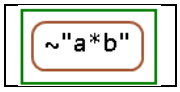


Figure 12 – Regular expression state

In this example, the object will be at the specified state if it is a string, and if the regular expression contained in the state can be matched with the value of the string that is stored in the object, for example, the string "`acb`".

An object can be at one state at any given time. Therefore while OPP might allow the definition of multiple states that can be satisfied at the same time (e.g., an object with value 4 and the two states $x < 5$ and $x < 10$), the execution of the program will result in an error if the interpreter detects an object that can satisfy more than one state.

3.2.3 Structural and Procedural Links

Links visually connect the nodes in an OPD, expressing relations between them. There are two primary kinds of links: structural links and procedural links.

3.2.3.1 Structural Links

Structural links define static relations between the nodes that they connect. They have one source and can have one or more targets. A pseudo-node (a visual node that represents the link but is not a node in the language) is used to collect all of the target-bound links, and these links are differentiated by the visual representation of this pseudo-node. There are two kinds of structural links: aggregation-participation and generalization-specialization.

3.2.3.1.1 Aggregation-Participation Link

An aggregation-participation link (aggregation for short) defines a static whole-part relation between the source of the link (the whole in the relation, the parent) and the target of the link (the part in the relation, the child). Aggregation links are represented by a filled-in (black) triangle pseudo-node, as shown in Figure 13. Aggregation links can have both objects and processes as sources and targets of the relation.

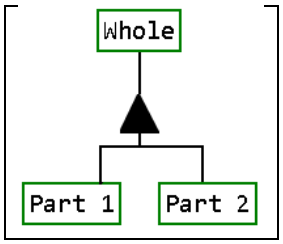


Figure 13 - Aggregation link

When used with objects, the programmer can abstract concepts by defining complex types. These complex types can later be used as regular objects, and their parts can be referenced

whenever needed in the program. An example of an aggregation relation is a bank account: it contains an account number, credit cards, checkbooks, current balance, etc. This is shown in Figure 14.

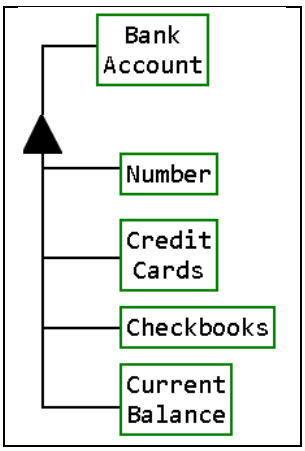


Figure 14 – Object aggregation example

When used with processes, the aggregation relationship shows which processes can be invoked by a process when it is executed. Furthermore, the aggregation relationship also shows the objects that are used by this process, both the parameters and the inner object. For example, the In-Zoomed process "Hello World" of the introductory example in Figure 1 can be represented using aggregation, as shown in Figure 15.

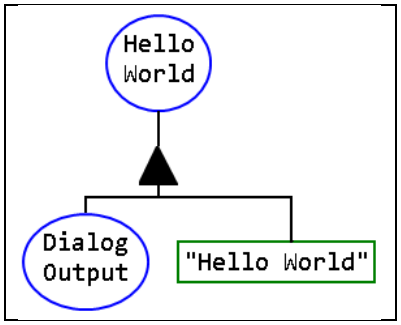


Figure 15 - Process aggregation

Object parts can be defined statically, as shown above, or dynamically, by using the OPP language's built-in processes defined in Section 4.3.4.3. When used dynamically, an object acts as a "Map" or "Dictionary" that stores key-value pairs.

3.2.3.1.2 Generalization-Specialization link

A generalization-specialization link (specialization for short) defines a static relation, which induces inheritance from the source of the link (the general, the parent) and the target of the link (the specialization, the child). Specialization links are represented by a blank (white-filled) triangle pseudo-node, as shown in Figure 16. Specialization links are only

allowed for the same kind of node in both ends – both parent and child must be either objects or processes.

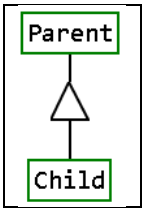


Figure 16 – Specialization link

Inheritance is defined by Dictionary.com [55] as "*something, as a quality, characteristic, or other immaterial possession, received from progenitors or predecessors as if by succession*". When used between objects, the specialization link defines that the child object inherits all the parts that are defined in the parent object. For example, suppose a person has the attributes ID and gender as shown in the OPD of on the left in Figure 17. A student is a specialization of a person with an added list of courses she or he is taking. A teacher is also a specialization of a person with an added list of courses he or she is teaching. Instead of defining the ID and the gender in both student and teacher, it is defined in the person, and then both student and teacher inherit this attribute from it. The visual representation for this is shown in Figure 17.

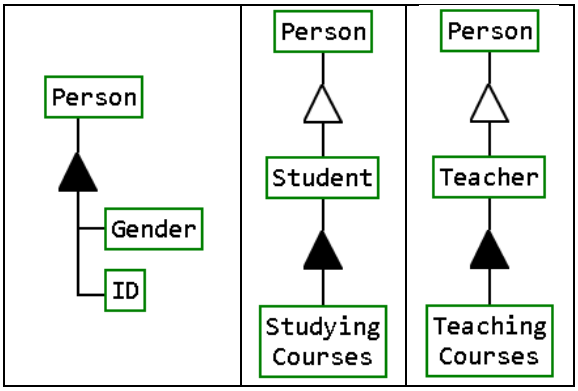


Figure 17 – Object specialization example

When used between two processes, a specialization relationship defines that the specialization (the child process) inherits and therefore has the same interface, i.e. the same set of parameters (as explained in Section 03.6) as the general (the parent process), and in addition, the specialization can have one or more parameters than the general does not have. The links connecting the parameters in the child process must be of the same kind as the links connecting the corresponding parameters in the parent process. In addition, if the parent process has defined pre-execution and post-execution processes (as explained in Section 4.2), these processes are automatically defined also in the child process. An example of process inheritance, shown in Figure 18, is the **Sorting** process, which can be defined generically: order a given set of objects by some internal object identifier, or specifically: **Numerical**

Sorting – order a given set of objects numerically, and **Lexicographical Sorting** – order a given set of objects lexicographically. The child processes will have the same set of parameters as the parent process: the input parameter **List**, and the output parameter **Sorted List**.

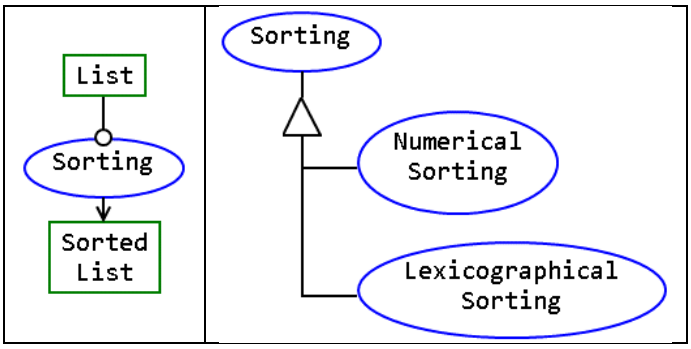


Figure 18 – Process specialization example

3.2.3.2 Procedural Links

Procedural links connect two nodes (source and target) in an OPD, denoting a dynamic relation between them. There are four kinds of procedural links: Agent, Instrument, Consumption, and Result.

1. Agent Link: An agent link defines a relation between an object or a state of an object and a process, where the source object is required by the process to start execution, but the value of the object is not passed to the process. If the source of the link is a state, the object must be in that state for the process to be able to execute. Agent links are represented by a connecting line between the two entities, with a closed black circle ("black lollipop") at the target process end of the link, as shown in Figure 19

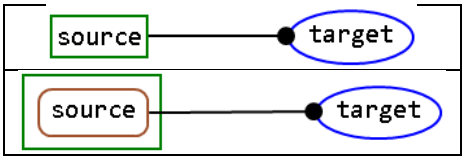
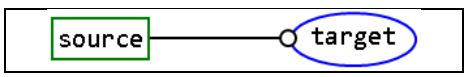


Figure 19 – Agent link with object and state source

2. Instrument Link: An instrument link defines a relation between an object (or state in an object) and an object or process, where the value of the source object is required by the target object or process. If the source of the link is a state, then the object is required to be at the specified state for the object or process to use it. Instrument links are represented by a connecting line between the two entities, with a black circle ("white lollipop") at the target process end of the link, as shown in Figure 20.



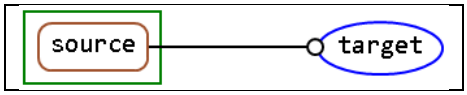


Figure 20 – Instrument link with object and state source

3. Consumption Link: A consumption link defines a relation between an object or a state of an object and a process, where the source object is consumed by the process. If the source of the link is a state, then the object is required to be at that state for the process to consume it. The semantics of consumption is that when the process is executed, the object ceases to have a value. Consumption links are represented by a connecting line between the two entities, with an arrow at the target process end of the link, as shown in Figure 21.

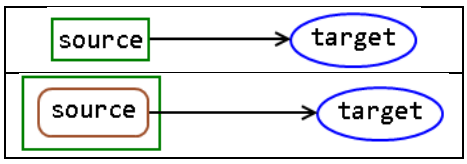


Figure 21 – Consumption link with object and state source

4. Result Link: A result link defines a relation between a process and an object, where the source process creates a new object. Result links are represented by a connecting line between the two entities, with an arrow at the target object end of the link, as shown in Figure 22.

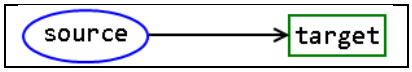


Figure 22 – Result link

Procedural links can be "decorated" with text at the center or at the target of the link, as shown in Figure 23. The center decoration can be any text that does not include the "," (comma) character. The target decoration is called a control modifier, and it is a comma-separated list of letters that can be "c" (for condition), "e" (for event), or both "ce".

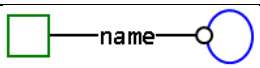
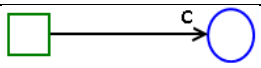
	
Instrument link with a center decoration	Consumption link with the target decoration "c" – a control modifier

Figure 23 – Procedural link decorations

3.3 SYSTEMS PROGRAMMING WITH OPP

An OPP system is composed of (1) a set of processes that the system exposes, (2) a set of global objects that are available to all the system processes, and (3) a set of data types that are used by the system. The processes that the system exposes show what the system can do. The aggregation of the states of all the global objects is the state of the system. The data types

facilitate the understanding of the system by giving semantics to the data flowing in the system.

All parts of the system are defined in the System OPD (called the System Diagram, or SD, in OPM). This System OPD is refined for each area of the system using other OPDs: the types are defined in Type OPDs, the interfaces of the external and internal processes are defined in Interface OPDs, and the executable semantics of the processes in the system are defined in In-Zoomed OPDs.

A good way to understand the definition of a system is by examining a real-world system example. An example of a software system, presented in the following subsections, is an editor of OPP programs. This system definition does not include the interface with which the user interacts with the system, as OPP is targeted primarily for server-side (or back-end) programming

3.4 The OPP Editor: A System OPD Example

The definition of a system starts with the System OPD. A system is defined as an abstract process, which consists of processes that implement the functionality of the system. Using the OPP editor example, the initial abstract process of this system is **OPP Editing**. It is customary in OPM to name processes with "ing" suffix (gerund form), and OPP follows this convention.

The **OPP Editing** process consists of a number of abstract processes: **User Managing**, **OPD Editing**, and **Model Managing**. Furthermore, the system stores a set of its **Users** and the set of all the **Models** in the system. This diagram is shown in Figure 24. Recall that the dashed contour around a process symbolizes that the process is abstract.

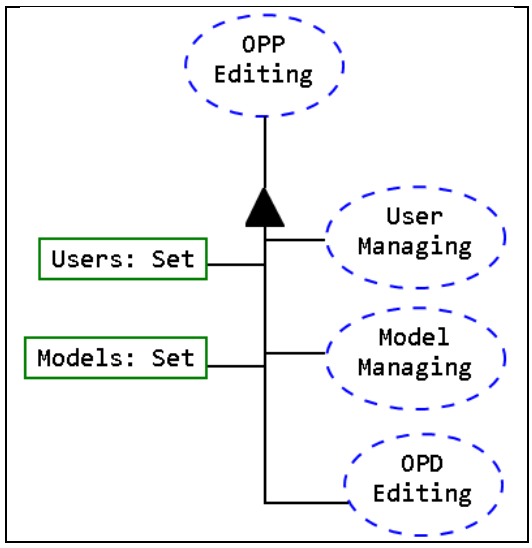


Figure 24 – OPP Editing System OPD

The full system definition requires for all leaf processes to be concrete, creating a tree-like hierarchy of processes. This hierarchy can be later used to define common execution properties for the processes in the hierarchy. Continuing with the example, the abstract **Model Managing** process is now refined as **Model Creating**, **Model Deleting**, **Model Renaming**, and **Model Duplicating**, as shown in Figure 25.

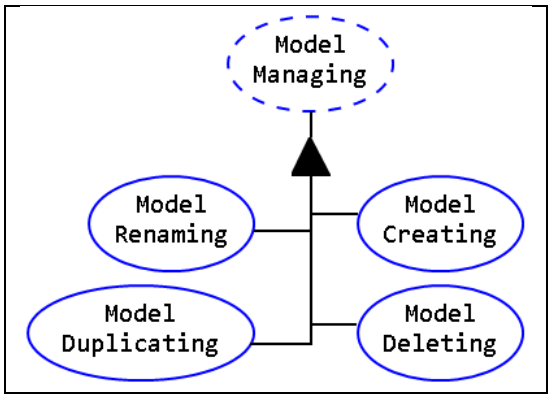


Figure 25 – Model Managing System OPD

The definition of a system can be done in a single OPD, yet it is both possible and recommended to separate the definition of the system into a hierarchy of system OPDs, each one defining a set of related functionality. In our example, the system OPD in Figure 25 is a child of the system OPD in Figure 24, in which **Model Managing** is defined

3.5 Type OPD

The OPP language is gradually typed [56], which means that the type of the object can be defined and then validated during compilation, or it can be left undefined, in which case types are validated at runtime so errors can occur during program execution.

As was shown in Section 3.2.2.1, the type of an object is defined in the label that is displayed on an object. There are three primitive types in OPP: **Number**, **String**, and **Any**. An object of type **Number** can contain values that are decimal numbers. An object of type **String** can contain any sequence of Unicode characters. An object of type **Any** can contain any value, either of a primitive type or of a type defined in a specific OPP program.

A new object type is defined in a Type OPD. A Type OPD has a name, which is the name of the type. For implementation simplicity, type names must start with an English letter, and after this contain English letters, spaces, and the characters '-' (dash) and '_' (underscore). A Type OPD must contain at least one object whose name is the same as the name of the OPD. In the type OPD, the developer can define the type by showing the parts of the new type, using Aggregation-Participation links. Continuing with the OPP editor example, Figure 26 shows an initial definition of a **OPP System** type. In this case, the name of the Type OPD is "OPP System" (not shown) which matches the **OPP System** object contained in it.

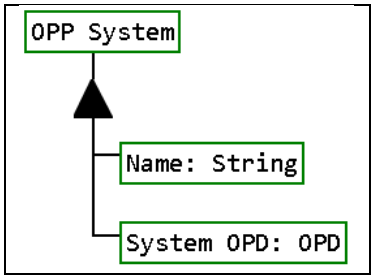


Figure 26 – First definition of the OPP System object type

3.5.1 Collection Objects: List and Set

OPP supports collections objects, each of which can contain any finite number of object instances. There are two types of collection objects in OPP: List and Sets. List is an ordered collection of object instances, while Set is an unordered collection of object instances with no duplicate objects, as defined by the object's identity (defined below). The syntax for defining object collection objects is shown in Figure 27.

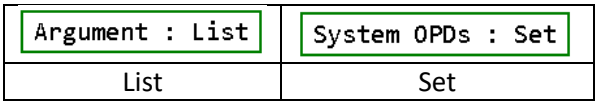


Figure 27 – List and Set objects

Manipulation of collection objects is done using processes that are described in Section 4.3. For the initial version of OPP, the language currently does not support typed collections, but this is a known requirement and will be added in the future.

Using collection objects, the definition of **OPP System** is extended to include multiple System OPDs, Type OPDs, Interface OPDs, and In-Zoomed OPDs, as is shown in Figure 28.

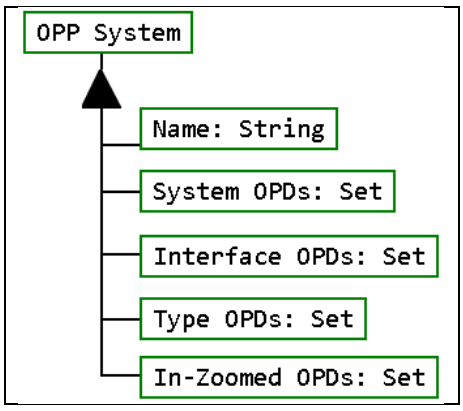


Figure 28 - OPP System definition

3.5.2 Object Inheritance

The second thing that can be defined in a Type OPD is object inheritance, which is done via the specialization relation. A specialized object (child) inherits all the defined parts of its general – another more general object (a parent or an indirect ancestor) – if they are

connected with a Generalization-Specialization link. An object can have multiple ancestors, giving rise to multiple inheritance. If a part is defined in more than one ancestor (either directly or indirectly via previous specializations) the OPP interpreter will check that this part is defined with the same type for all the ancestors. If this is not the case, the type will not be accepted by the interpreter at runtime, and execution of the system will be halted with an error message.

Figure 29 shows an example of object inheritance for the OPP Editor. Conceptually, OPP objects and processes are similar in many aspects: they both have names, they have a location in the OPD, they have a width, height, etc. Capitalizing on OPM, the definition of the shared properties can be done in a type called **Thing**, from which both **Object** and **Process** inherit.

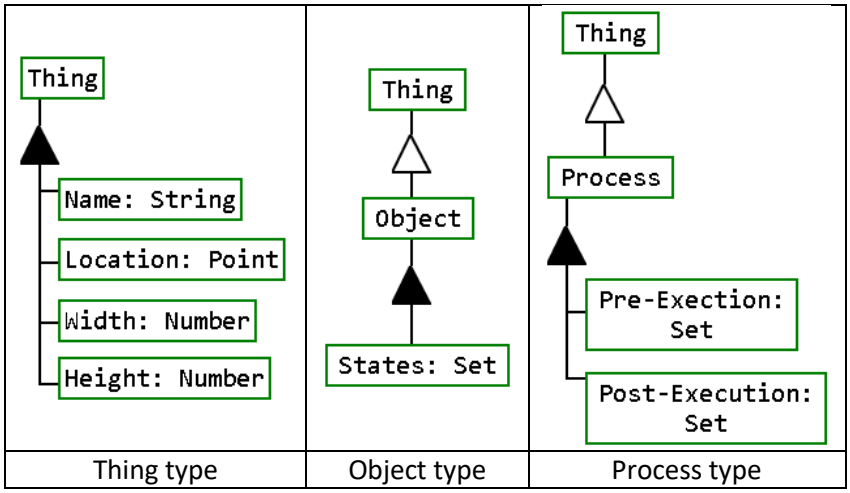


Figure 29 – Definition of Thing, Object and Process types, using object inheritance

3.5.3 Object Identity

Each OPP object has a built-in identity, which is unique to the object and does not change, and can be accessed through the object's **Id** part, as shown in Figure 30. For object of basic types such as **Number** and **String**, the **Id** of the object is the same as the value of the object. When using Sets, the identity of an object is used to maintain the collection duplicate-free. Objects in OPP are immutable, meaning that every object that is created is a new object, even if it has the exact same information as an existing object.

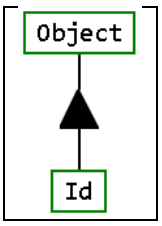


Figure 30 – Object identity

The default identity of an object can be overridden by defining a new part of the object as the identity. This is done by adding the text "(Id)" after the part definition. The Type OPD shown in Figure 31 exemplifies this with the type **Car** which has the custom identity **License**.

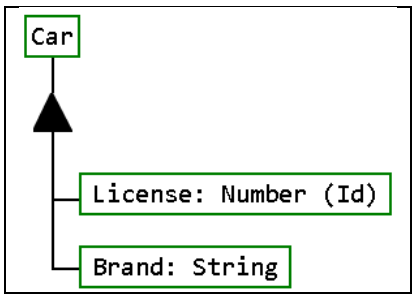


Figure 31 – Object with custom identity

3.6 INTERFACE OPD

An interface OPD defines the parameters of a process, and the definition of pre- and post-processes to be executed before (pre-execution processes) and after (post-execution processes) the process is executed. Interface OPDs can be thought of as mini-APIs (Application Programming Interface).

A simple example of an Interface OPD is shown in Figure 32. It contains a central process, the process for which the interface is defined, as well as procedural links to objects which define the parameters of the process. In this example, **Adding** has three parameters, two incoming instrument – **a** and **b** of type **Number**, and one outgoing result – **c** of type **Number**.

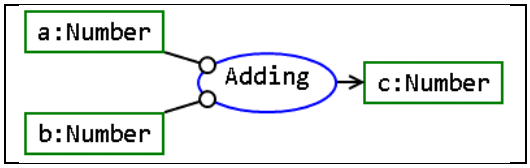


Figure 32 - Basic interface OPD

A process definition can be augmented by adding pre-executing processes and post-executing processes. Processes that are defined as pre-executing processes are executed before the process is executed, and those defined as post-execution processes are executed after the process is executed.

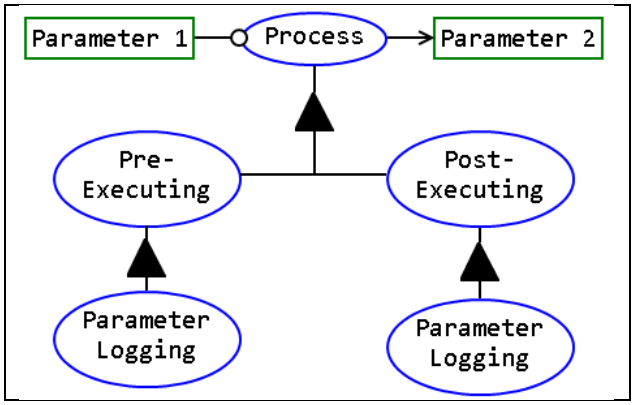


Figure 33 – Pre-executing and Post-executing processes

The OPD in Figure 33 extends the definition of **Process** from Figure 32 by adding the **Parameter Logging** process as both pre-executing and post-executing process. **Parameter Logging** logs the value of the parameters that were passed to **Process** and from **Process** to a pre-defined output. This is useful for debugging purposes. While not shown in the example, a process can have more than one pre-executing and post-executing process. However, if there is more than one, there is no guarantee on the order in which they will be executed.

3.6.1 Process Inheritance

A process can be defined as a specialization of another process by using the generalization-specialization link, and thereby inherit from the general, ancestor process. Process inheritance is defined as follows:

- The inheriting process includes all the pre-executing processes and post-executing processes from all its ancestor processes. If a process is defined more than once in the pre-executing processes or in the post-executing processes, it will be executed only once.
- The inheriting process inherits all the parameters defined for the ancestor process, and they are inherited with the same names. More parameters can be added to the inheriting process in addition to the ones defined in the ancestor process.

An example of using process inheritance, shown in Figure 34, is to define processes that have common prerequisites, such as authentication. We first define a common abstract process **Authentication Requiring**, which has **Authenticating** as a pre-executing process. Then we define the process **Ordering** as inheriting from **Authentication Requiring**, making **Authenticating** one if its pre-executing processes through the inheritance mechanism.

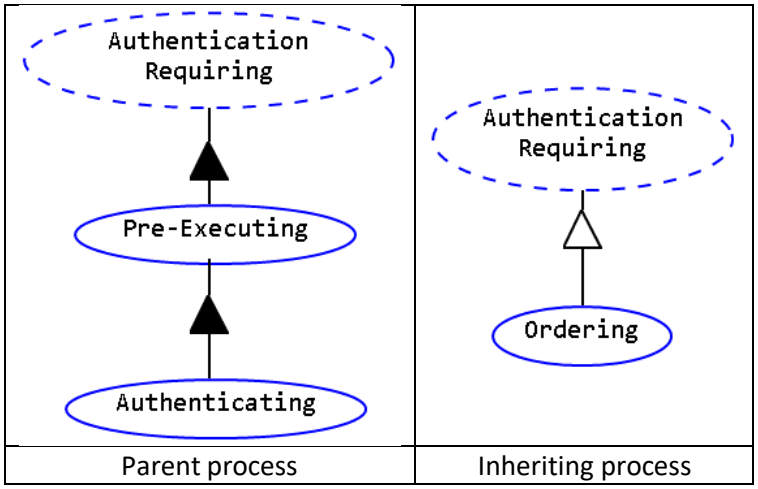


Figure 34 – Process inheritance

Pre-executing processes and post-executing processes may want to share data with the processes that defines them. Section 4.1 includes an explanation on how data is accessed by pre-executing and post-executing processes, and how data can be shared between them and the main executing processes.

3.7 In-Zoomed OPD

An In-Zoomed OPD is the executable code of the language. It describes a set of execution steps using procedural and event-driven constructs. We provide an informal description of how In-Zoomed OPDs are executed, followed by a formal definition.

3.7.1 Informal Executable Semantics

An In-Zoomed OPD consists of a main process (the In-Zoomed process), which can contain processes, objects, and links that connect them. Objects are data stores: values can be stored in them and read from them. Links define how values are passed to or from processes, or between objects. Objects provided to the In-Zoomed process or yielded by it – the Parameters of the process – are located outside that In-Zoomed process.

Continuing with the OPP editor example, Figure 35 shows the In-Zoomed process **Command Executing**, which takes a command [57] that is sent from the editor, and applies it to the current OPD to create a new OPD. The name of the process in the OPP Editor is shown in the editing window (not shown here), and not in the In-Zoomed process ellipse to reduce the number of visual elements shown in it, as it can be understood by the developer from the context.

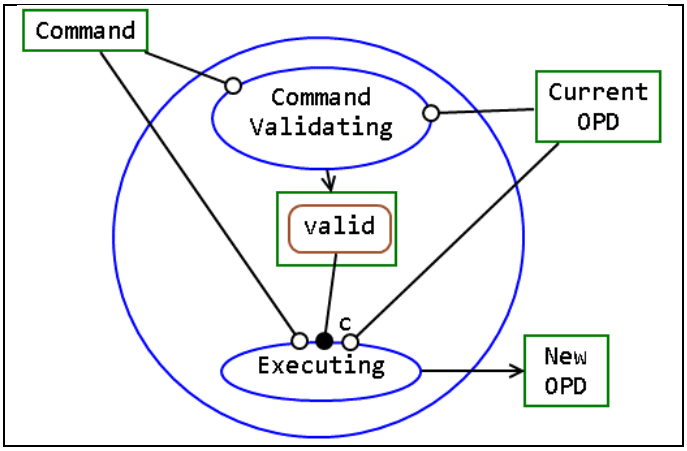


Figure 35 - In-Zoomed OPD: Command Executing

The large ellipse is the In-Zoomed process. This process has:

- Three parameters: **Command**, **Current OPD**, and **New OPD**
- One internal object with one state named **valid**
- Two inner processes: **Command Validating** and **Executing**.

The execution of an In-Zoomed OPD is initiated by the interpreter based on an external event, such as invocation by another process. When a process is invoked, it is assumed that all its input parameters (as defined in its API or in the process itself) have values.

Following the timeline OPM principle [58], execution of an In-Zoomed OPD is done from top to bottom. Hence, the first process to be invoked is **Command Validating**. This process requires the values of the parameters **Command** and **Current OPD**. **Command Validating** is executed, and its result is set in the anonymous object located inside the In-Zoomed process. Following the timeline (top-to-bottom process order), the process **Executing** is invoked next. Since **Executing** is connected to the state **valid** of the anonymous object with an agent conditional link (which will be defined later), it will be executed only if the object is in this state, otherwise it will be skipped. When the process finishes, the interpreter will set its result in the parameter object **New OPD**. If it is skipped, this object will have no value. After **Executing** finishes, the entire In-Zoomed **Command Executing** process finishes. If this process was invoked by another process **P**, **P** will get an event telling it that **Command Executing** has finished.

3.7.2 Formal Execution Semantics

Having exemplified the execution of an In-Zoomed process informally, we turn to define the execution semantics formally.

3.7.2.1 Definitions

An In-Zoomed OPD is a tuple consisting of a set of object parameters, object variables, inner processes, and links:

$$OPD_{in-zoomed} = (O_{par}, O_{var}, S, P, L)$$

Where

- O_{par} is the set of the input and output parameters of the In-Zoomed process, which are the objects located outside the In-Zoomed process ellipse.
- O_{var} is the set of variables of the In-Zoomed process, which are the objects located inside the In-Zoomed process ellipse.
- S is the set of all the states defined in $O_{par} \cup O_{var}$. The function $object(s \in S)$ returns the object that contains state s.
- P is the set of processes located inside the In-Zoomed process ellipse. In an In-Zoomed OPD no processes can be located outside the In-Zoomed process ellipse.
- L is the set of all procedural links in the OPD.

A link is a tuple

$$l = (s, t, kind, mod)$$

Where

- s and t are the source and target entities of the link, respectively.
- $kind \in \{Agent, Inst, Cons, Res\}$ is the link kind, where the symbols inside the set stand for Agent, Instrument, Consumption, and Result, respectively. Structural links do not affect execution therefore are not taken into account in this set.
- mod is the set of control modifiers applied to the link. The OPP control modifiers are $'c'$ for condition link and $'e'$ for event link. None, one, or two control modifiers can be added to Agent, Instrument, and Consumption links.

To simplify the notation, we will use l_s, l_t, l_{kind}, and l_{mod} to identify the source, target, kind, and modifier set of the link, respectively.

Based on these definitions, we analyze the OPD in Figure 36 as an example.

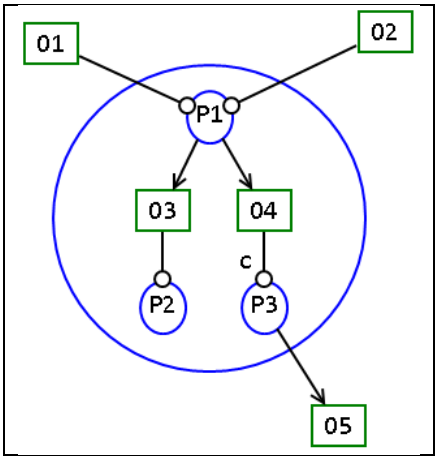


Figure 36 - In-Zoomed process components

For simplicity, we use the names of the objects as their identifiers. While this is correct in this OPD, there can be more than one object in an OPD with the same name. Moreover, an object is not required to have a name, so this will not work in all cases.

- $O_{par} = \{O1, O2, O5\}$
- $O_{var} = \{O3, O4\}$
- $P = \{P1, P2, P3\}$
- $L = \{(O1, P1, Inst, \emptyset),\ (O2, P1, Inst, \emptyset),\ (P1, O3, Res, \emptyset),\ (P1, O4, Res, \emptyset),\ (O3, P2, Inst, \emptyset),\ (O4, P3, Inst, \{c\}),\ (P3, O5, Res, \emptyset)\}$

Execution of an In-Zoomed process is done in three steps:

1. **Incoming argument transfer**: the value of the parameters provided by the calling process is stored in the corresponding (matching) variables of the In-Zoomed process.
2. **Process execution**: the processes contained in the In-Zoomed process are executed based on the OPP execution model.
3. **Outgoing argument transfer**: the values of the variables matching the outgoing parameters are stored in the matching objects of the calling process.

The following sections will describe each of this steps and its sub-steps.

3.7.2.2 Incoming and Outgoing Argument Transfer

When an OPP process is invoked, either directly by an external user operation or by another process, all its incoming parameters must have values. These values are the arguments that are passed in the invocation.

The incoming parameters of a process are the objects that are connected to the process with an instrument or consumption link (agent links are used only to control process execution; therefore, they are not parameters of a process). The outgoing parameters of a process are the

objects that are connected to the process with a result link. An example of this is shown in the Interface OPD in Figure 37.

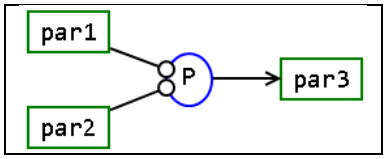


Figure 37 – Process parameters

Process **P** has three parameters: **par1**, **par2**, and **par3**. The first two, **par1** and **par2**, are incoming parameters, while **par3** is an outgoing parameter.

The Interface OPD can be derived from the In-Zoomed OPD. Each object outside the In-Zoomed ellipse is a parameter to the process. Parameters connected to processes inside the In-Zoomed ellipse with an incoming procedural link are connected to the Interface process with an instrument link. Parameters yielded by processes inside the In-Zoomed ellipse are connected to the Interface process with a result link. If and object is connected in the In-Zoomed process to both an incoming and an outgoing procedural link, both links are also shown in the Interface OPD.

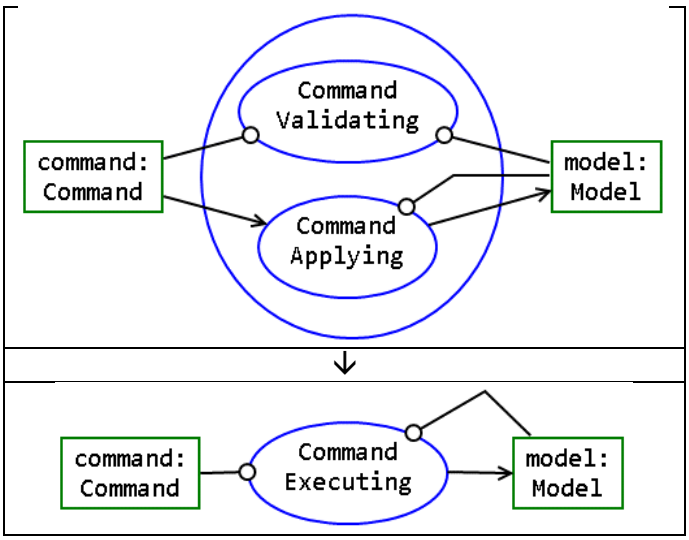


Figure 38 – In-Zoomed OPD to Interface OPD transformation

Interface OPDs use only instrument links because of the immutability of OPP objects. When an object is given to a process as a parameter, it is passes by value – a copy of the object is created to be used by the process. Because of this, it is not possible to define consumption from the perspective of an interface. Consumption can be used by the developer in an In-Zoomed process by replacing an instrument link with a consumption link. In this case, the object will be copied to the consuming process and will be deleted from the In-Zoomed process.

We decided to make OPP objects immutable because of the parallelism that is built into OPP. Mutable objects are inherently not thread-safe, which would make the language more complex to implement and to use.

When a process is invoked, a value is assigned to each one of its incoming parameters. These values are called the arguments of the process. The value of each of the incoming arguments of the process is copied to the corresponding incoming parameter object defined in the process being invoked. When the process finishes executing, the value of the outgoing parameter is copied back to the argument defined in the called process. Using the Interface OPD from Figure 37, process **P** is now invoked by another process, as shown in Figure 39.

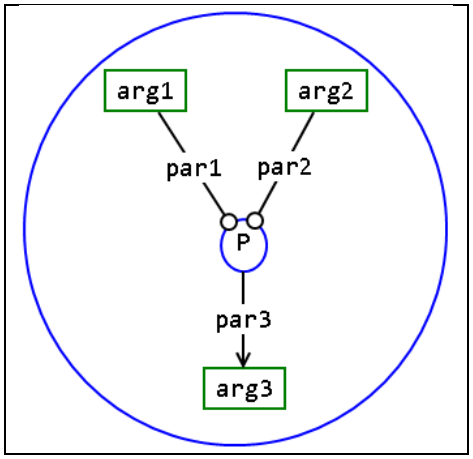


Figure 39 – Arguments passing with parameter names

Process **P** is given two arguments, **arg1** and **arg2**, as the values for parameters **par1** and **par2**, respectively. When **P** finishes executing, the value of parameter **par3** will be copied to **arg3**.

In the example shown in Figure 39, the interpreter is provided with an explicit mapping between the arguments given by the calling process and the parameters of the process. This is done by stating the name of the parameter in the link that connects the argument with the called process. While necessary in some cases, having to write the name of the parameter in all the links is cumbersome and often redundant. To avoid this when possible, the interpreter has a set of rules which it applies to match arguments and parameters separately for each type of incoming and outgoing link of the process. These rules are specified below and are applied sequentially in the following order:

1. **Named arguments**: When the link that connects the argument object to the invoked process has text located at its center, this text is used by the interpreter as the name of the parameter in the invoked process. If the parameter object is typed, the type of the argument object must match the type of the parameter, or inherit the type from the parameter.

2. **Named objects and parameters matching**: If the name of the argument object matches the name of one of the parameters of the invoked process, and this parameter was not matched by the previous rule with another object, then the named object will be used as the parameter. As in the previous rule, if the parameter object is typed, the type of the argument object must match the type of the parameter or inherit the type from the parameter.
3. **Type matching**: if there are still parameters that do not have a matching argument, and there are available arguments that have not been matched, the interpreter will try to match for each available parameter that has a type an argument object that is of the same type or of an inherited type.
4. **Everything else**: remaining arguments are matched to remaining parameters arbitrarily.

If after applying the rules above there are parameters without a value, the interpreter will not execute the invoked process and execution of the currently executing process will stop.

Figure 40 is an example of the parameter matching rules.

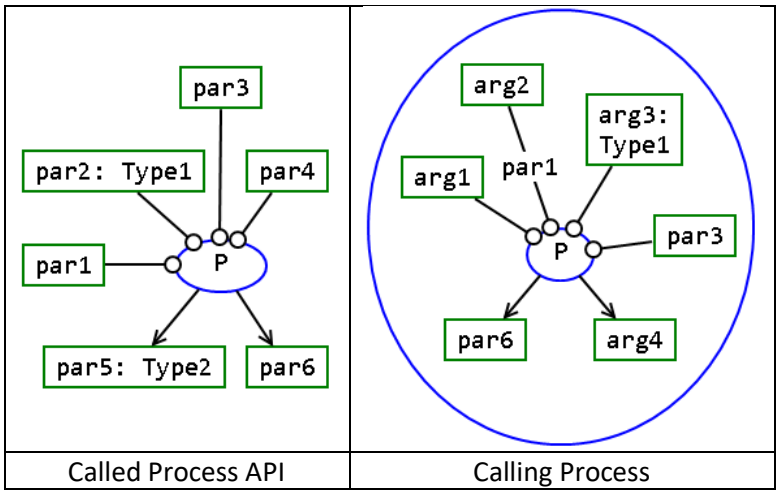


Figure 40 – Argument to parameter matching

Process **P** has four incoming parameters that are connected with an instrument link: **par2**, which is of type *Type1*, **par1**, **par3**, and **par4**, which have no defined type. Process **P** has two outgoing parameters (which can only be connected by a result link), **par5**, which is of type *Type2*, and **par6**, which has no defined type.

Starting with the given incoming arguments when **P** is invoked, the first match is done using rule 1, the **Named arguments** rule, which checks for named arguments. Since the argument **arg2** is connected with a link decorated with the text *par1*, it will be used as the value for parameter **par1**. As there are no more named arguments, rule 2, **Named objects and parameters matching**, is applied, searching for objects that are named as a parameter. Since **par3** is the name of an argument that is the same as the name parameter **par3**, it is used

as its value. Now, rule 3, **Type matching**, is applied, searching for type matches. We find that **arg3** matches the type of **par2**, since both are of *Type1*, so they are matched, and the value of parameter **par2** is assigned to **arg3**. Lastly, **arg1** is matched with **par4** as these are the only parameter-argument pair left to match. The same set of rules is applied to the outgoing arguments, so that when the matching process finishes, the value the parameters **par5** is assigned to the value of argument **arg4**, and the value of the parameter **par6** is assigned to the value of the argument **par6**.

3.7.2.3 Top-Down Execution

The default order of executing processes inside an In-Zoomed process is based on the vertical location of the process relative to each other, based on the top-most point of the process ellipse. Processes are executed from top to bottom, using a program counter (PC). They are executed asynchronously, so multiple processes can be executed in parallel. When the In-Zoomed process starts executing, PC is initialized to the y coordinate of the top-most process inside the In-Zoomed process ellipse. An iterative loop is performed, and in each iteration, all processes whose top-most coordinate is at the PC are sent to execution. The interpreter then waits for a process to finish executing and tries to increment the PC to the top-most coordinate of the next process in the diagram below the PC, but this can only be done if the bottom-most coordinate of all executing processes remains above this point. In other words, the PC waits for all the processes whose bottom-most coordinate is above the next process to finish before invoking that process.

Formally, using a coordinate system with $y = 0$ at the topmost point in the In-Zoomed process ellipse and increasing downwards, let:

- $p_i \in P$ be a process,
- $top(p_i)$ and $bottom(p_i)$ be the top and bottom coordinates of process p_i, respectively,
- $PC \in \mathbb{N}$ be the program counter, and
- $P_{executing}$ be the set of processes that are currently executing.

We define $nextPC$ as the next coordinate that can be given to PC. This is also the top-most coordinate of the next process below the current PC, and it is defined using the following equation:

$$nextPC = \begin{cases} \infty & , PC = \max(\{top(p) | p_i \in P\}) \\ \min(\{top(p) | p \in P \wedge top(p) > PC\}) & , PC < \max(\{top(p) | p_i \in P\}) \end{cases}$$

We also define a Boolean helper function $canAdvance(PC)$ that returns $false$ if the PC cannot be incremented because there is at least one executing process whose bottom-most coordinate is above PC, which means that the PC cannot be incremented, and $true$ otherwise, which means that the PC can be incremented. Formally:

$$canAdvance(PC) = \begin{cases} false & \exists p \in P_{exec}, bottom(p) < PC \\ true & otherwise \end{cases}$$

Using the definitions above, the top-down execution of an In-Zoomed process is performed according to Algorithm 1 – **OPP Execution v1**. Processes in OPP are executed asynchronously, and inserting the process into P_{exec} means that the process is executed.

	OPP Execution v1
1	$PC \leftarrow \min(\{top(p) \mid p \in P\}), P_{exec} \leftarrow \{p \in P \wedge top(p) = PC\}$
2	While $P_{exec} \neq \emptyset$
3	Wait for $p_i \in P_{exec}$ to finish
4	$P_{exec} \leftarrow P_{exec} \setminus \{p_i\}$
5	$PC_{cand} \leftarrow nextPC$
6	If $canAdvance(PC_{cand})$ then
7	$PC \leftarrow PC_{cand}$
8	$P_{exec} \leftarrow P_{exec} \cup \{p \in P \wedge top(p) = PC\}$
9	End If
10	End While

Algorithm 1 – Top-Down Process Execution

We demonstrate the execution of Algorithm 1 using the In-Zoomed process shown in Figure 41, where $top(p1) = top(p2) < top(p3) = top(p4) < top(p5) < top(p6)$.

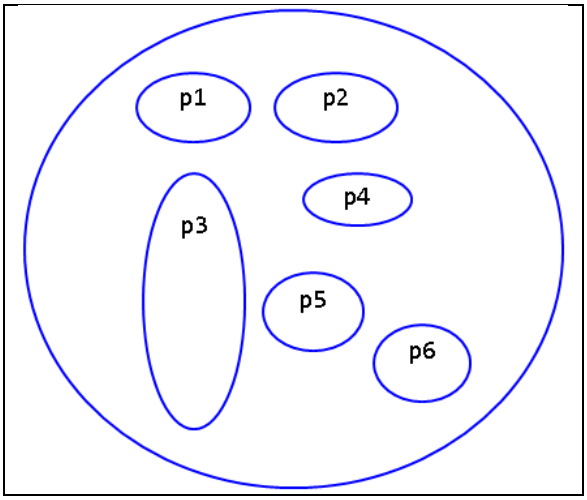


Figure 41 - In-Zoomed process for top-down execution

PC is first set to $top(p1)$ and $P_{exec} \leftarrow \{p1, p2\}$. The iterative loop is entered, and let's assume $p2$ finishes executing first. PC_{cand} is now $top(p3)$, but since $botom(p1) < PC_{cand}$ and $p1 \in P_{exec}$, the PC cannot be lowered and a new iteration is started. Now $p1$, the only executing process, finishes. Since $P_{exec} = \emptyset$, $PC \leftarrow PC_{cand}$ which is $top(p3)$, and $P_{exec} \leftarrow \{p3, p4\}$. In the next iteration, assume that $p4$ finishes first. In this case, $P_{cand} = top(p5)$ and this is above $bottom(p3)$, so even though $p3$ is still executing, the PC can be updated to

$top(p5)$, and $p5$ can be executed. In the next iteration $p6$ will be executed, and when all the executing processes finish, the execution of the In-Zoomed process finishes as well.

3.7.2.4 Data Dependencies

The data in an In-Zoomed OPD is stored in objects. An object can get a value in the following ways:

- Objects defined outside the In-Zoomed OPD are assigned values by the calling process when the process is invoked.
- An object can be initialized using value literals, as shown in Section 3.2.2.1.
- When a called process finishes executing, it may yield one or more results that are stored in objects.
- An object can be connected to another object using an instrument link, so when the value of the source object changes, the value of the target object changes to the value of the source object.

As previously stated, OPP objects are immutable, which means that when an object is passed as an argument to a process, the value of the object is copied to a new object created in the called process. The same happens when a process finishes executing – the value of any outgoing parameter is copied to the corresponding target object (argument) in the calling process. The same is valid for objects connected with instrument links: the value of the source of the link is always copied as the value of the target of the link.

The execution of processes inside an In-Zoomed process is dependent on the objects they use in their execution. A process requires an object if it has an incoming procedural link that starts at the object or at one of its states and ends at the process. Let $dep(p)$ to be the set of all entities on which process p depends:

$$dep(p) = \{o \in (O_{par} \cup O_{var}) \wedge l \in L \wedge l_s = o \wedge l_t = p\} \cup \{s \in S \wedge l \in L \wedge l_s = s \wedge l_t = p\}$$

A process can be executed when all its dependencies are ready. The readiness of a dependency depends on its type as follows.

- If the dependency is on an object (i.e. there is an incoming link from an object to the process being executed), the dependency (object in this case) is ready when the object has a value (any value).
- If the dependency is on a state (i.e. there is an incoming link from an object's state **S** to the process being executed), the dependency (state in this case) is ready when the object is at state **S**, as defined in Section 3.2.2.3.

Based on this, we define $isReady(e)$ to return $true$ if the entity is ready, $false$ otherwise.

When a process is not ready, its execution is postponed until it is ready. At the same time, the execution algorithm now must take into account that it cannot lower the PC below all

executing or waiting processes. This means the definition of function $canAdvance(PC)$ has to be updated, and is now as follows:

$$canAdvance(PC) = \begin{cases} false & \exists p \in (P_{exec} \cup P_{waiting}), bottom(p) < PC \\ true & otherwise \end{cases}$$

Based on this update, the execution algorithm is updated in Algorithm 2 below to **OPP Execution v2**, which includes the new set of waiting processes and their management during execution. For better readability, we do not add to the algorithm steps that handle parameter/argument transfer from/to the processes that are executed, and to/from the In-Zoomed process being executed, as these steps are straightforward.

	OPP Execution v2
1	$PC \leftarrow \min(\{top(p) \mid p \in P\}), P_{wait} \leftarrow \{p \in P \wedge top(p) = PC\}$
2	$P_{exec} \leftarrow \{p \in P_{wait} \wedge isReady(p)\}$
3	$P_{wait} \leftarrow P_{wait} \setminus P_{exec}$
4	While $P_{exec} \neq \emptyset$
5	Wait for $p \in P_{exec}$ to finish
6	$P_{exec} \leftarrow P_{exec} \setminus \{p\}$
7	$PC_{cand} \leftarrow nextPC$
8	If $canAdvance(PC_{cand})$ then
9	$PC \leftarrow PC_{cand}$
10	$P_{exec} \leftarrow P_{exec} \cup \{p \in P \wedge top(p) = PC \wedge isReady(p)\} \cup \{p \in P_{wait} \wedge isReady(p)\}$
11	$P_{wait} \leftarrow P_{wait} \setminus P_{exec} \cup \{p \in P \wedge top(p) = PC \wedge \neg isReady(p)\}$
12	Else
13	$P_{exec} \leftarrow P_{exec} \cup \{p \in P_{wait} \wedge isReady(p)\}$
14	$P_{wait} \leftarrow P_{wait} \setminus P_{exec}$
15	End If
16	End While
17	Error if $P_{wait} \neq \emptyset$

Algorithm 2 – In-Zoomed process execution with dependencies

The example OPD in Figure 42, where $top(P1) = top(P2)$, demonstrates how the new algorithm works.

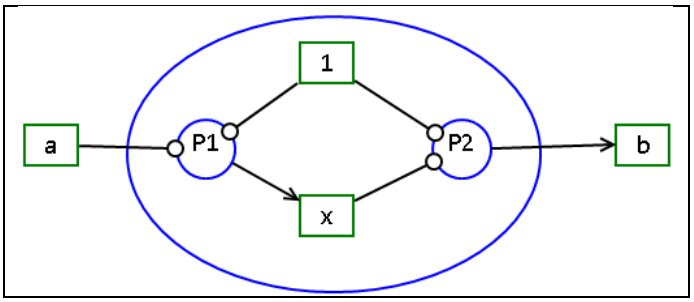


Figure 42 – Example In-Zoomed OPD with data dependencies

In line 1, $PC \leftarrow top(P1)$ and $P_{wait} \leftarrow \{P1, P2\}$. At this stage, variable x inside the In-Zoomed process does not have a value, so $P2$ is not ready, therefore, at line 3 we have $P_{exec} =$

$\{P1\}$ and $P_{wait} = \{P2\}$. After $P1$ finishes executing, there is a value in x. Line 7 gives $PC_{cand} \leftarrow \infty$, but in Line 8 we get that the PC cannot advance since there is a waiting process above the new PC candidate. Hence, in Line 13 the interpreter searches for new ready processes inside the waiting set and finds that $P2$ is ready. Therefore, at line 15 we have $P_{exec} = \{P2\}$ and $P_{waiting} = \emptyset$. A new iteration starts, $P2$ finishes executing, and the whole process execution finishes.

3.7.2.5 Condition Links

So far, if a process dependeds on an entity that was not ready, the process would be added to the waiting set and executed when the entities on which it depends became ready. Condition links provide a new degree of control of execution. They enable replacing wait semantics with skip semantics: If a process has a dependent entity that is connected with a condition link, and the dependent entity is not ready, the process is skipped instead of being waited upon to become ready. This means that condition links can be used as control statements like *if-then* or *switch* in textual programming languages. The condition link control modifier can be used in Agent, Instrument, and Consumption links, and is represented by the letter "c" added to the target end of the procedural link, as shown in Figure 43.

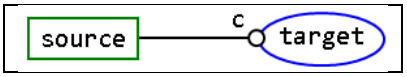


Figure 43 – Instrument link with conditional link modifier

To include condition links in the execution algorithm, using the notation $l_{mod}(e,p)$ to denote the set of control modifiers that are applied to the link connecting entity e (object or state) and process p, we define the function $isSkipped$ as follows:

$$isSkipped(p) = \begin{cases} true & e \in dep(p) \wedge 'c' \in l_{mod}(e,p) \wedge \neg isReady(e) \\ false & otherwise \end{cases}$$

In the previously defined algorithm (Algorithm 2, line 10), processes that are executed in each iteration are always at the PC. With conditional links, it may be the case that all processes at the PC are skipped, which means that the PC must be lowered again to find processes to execute, and this can happen multiple times. To handle this case, in Algorithm 3 we define a recursive function, called $executeNextProcesses$, which updates the sets $P_{waiting}$ and $P_{executing}$, as well as the PC as it performs the search.

	$executeNextProcesses$
1	$P_{cand} \leftarrow \{p \in P \wedge top(p) = PC\}$
2	$P_{skipped} \leftarrow \{p \in P_{cand} \wedge isSkipped(p)\}$
3	If $P_{skipped} = P_{cand}$ then
4	$PC \leftarrow nextPC$
5	$NextProcesses$
6	Else
7	$P_{exec} \leftarrow \{p \in P_{cand} \wedge p \notin P_{skipped} \wedge isReady(p)\} \cup \{p \in P_{wait} \wedge isReady(p)\}$
8	$P_{wait} \leftarrow \{p \in P_{cand} \wedge p \notin P_{skipped} \wedge \neg isReady(p)\} \cup \{p \in P_{wait} \wedge \neg isReady(p)\}$
9	End If

Algorithm 3 – executeNextProcesses function

In Algorithm 4, we define the function $executeCurrentProcesses$ as a function that is performed in case the PC is not advanced.

	$executeCurrentProcesses$
1	$P_{exec} \leftarrow P_{exec} \cup \{p \in P_{wait} \wedge \neg isSkipped(p) \wedge isReady(p)\}$
2	$P_{wait} \leftarrow \{p \in P_{wait} \wedge \neg isSkipped(p) \wedge \neg isReady(p)\}$

Algorithm 4 – executeCurrentProcesses function

Using these functions, we can now update the execution algorithm to include interpretation of conditional links, as shown in Algorithm 5.

	OPP Execution v3
1	$PC \leftarrow \min(\{top(p) \mid p \in P\}), P_{waiting} \leftarrow \{p \in P \wedge top(p) = PC\}$
2	$P_{exec} \leftarrow \{p \in P_{wait} \wedge isReady(p)\}$
3	$P_{wait} \leftarrow P_{wait} \setminus P_{exec}$
4	While $P_{exec} \neq \emptyset$
5	Wait for $p \in P_{exec}$ to finish
6	$P_{exec} \leftarrow P_{exec} \setminus \{p\}$
7	$PC_{cand} \leftarrow nextPC$
8	If $canAdvance(PC_{cand})$ then
9	$executeNextProcesses$
12	Else
13	$executeCurrentProcesses$
14	End If
15	End While
16	Error if $P_{waiting} \neq \emptyset$

Algorithm 5 – In-Zoomed process execution with conditional links

In what follows, we demonstrate how Algorithm 5 works on the In-Zoomed OPD in Figure 44. This OPD gets two numbers and yields "low" if the sum is less than 5, and "high" if the sum is equal to or greater than 5. For easier reference, the process **Object Copying** on the left hand side will be called $OC1$ and the one on the right – $OC2$.

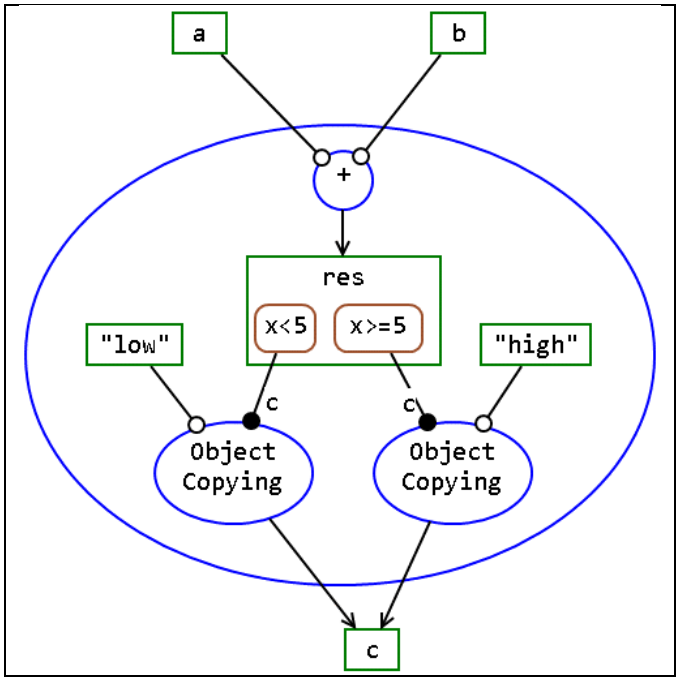


Figure 44 – Example In-Zoomed OPD with conditional links

Let us execute the OPP program in Figure 44 using Algorithm 5, assuming $a = 2$ and $b = 4$. Initially, $PC \leftarrow top(+)$ and $P_{waiting} \leftarrow \{+\}$ (line 1). Since the process was invoked, all the arguments required to start $+$ are ready, so $P_{exec} \leftarrow \{+\}$ and $P_{wait} \leftarrow \emptyset$ (lines 2-3). Entering the loop, the interpreter waits for $+$ to finish and removes it from P_{exec} (lines 4-6), also setting $res = 6$. Since the top-most points of both **Object Copying** process ellipses are at the same height, let us assume that $PC_{cand} \leftarrow top(OC1)$. Since $P_{exec} \cup P_{wait} = \emptyset$, the PC can advance (lines 7-8). At this stage, the function $executeNextProcesses$ is invoked (line 9). $P_{cand} \leftarrow \{OC1, OC2\}$, and since $res = 6$, state $n < 5$ is not ready and $n \geq 5$ is ready, so $P_{skipped} \leftarrow \{OC1\}$, leaving $P_{exec} \leftarrow \{OC2\}$ and $P_{wait} \leftarrow \emptyset$. The execution algorithm continues and waits for $OC2$ to finish, yielding the value "$high$" in object **c** as the result of the execution.

3.7.2.6 Event Links

The last way in which the execution of OPP can be modified is by using event links. Any change to an object, such as assigning a value to it, changing its value, or removing a value from it (using a consumption link) is defined as an event. Event links break the default top-down execution model of the interpreter and allow the user to define a specific process to be executed when such an event occurs. An even link is akin to *Jump* and *Goto* statements in procedural programming, and can be used to create loops, and more generally, event-driven programs. The event link control modifier can be used in Agent, Instrument, and Consumption links and is represented by the letter "**e**" added to the target end of the procedural link, as shown in Figure 45.

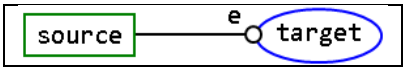


Figure 45 – Instrument link with an event link control modifier

Every time a process finishes and the values yielded by this process are assigned to objects in the In-Zoomed OPD, the interpreter checks if there are any event links that originate at any one of these objects. If this is the case, it sets its execution mode to event mode and executes the target process of this link without delay. While still in event mode, the interpreter waits for all the executing processes to finish, after which it returns to the "business as usual" top-down execution mode, with the PC at the top of the process that is directly below the most recently invoked process. In case the event link originates from a state, the event is triggered only if that state is ready, i.e., the value of the object matches the state, as defined in Section 3.2.2.3.

Because there is only one PC in OPP, only one event can be triggered after the execution of a process. If, at any stage, the interpreter detects that multiple events are triggered after a process finishes executing, it issues an error and stops the execution of the system.

Event links can also be used to trigger the execution of one process when a process finishes. This is done by connecting the two processes with an agent link that has an event modifier, as shown in Figure 46. The behavior of the interpreter in this case is the same as if an object yielded by the source of the link triggered an event that is directed at the target of the link.

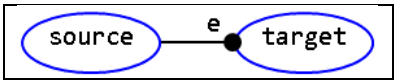


Figure 46 – Process invocation using agent link with event modifier

To handle event links, we define the function $findInvoked(p)$, which is called after a process finishes executing. This function goes through all the result entities of process p in search of outgoing event links, and for each one found, it also checks if the target process of the event link should be skipped, in which case the event is disregarded. This can happen if a process has an incoming conditional link from an object that is not ready.

	$findInvoked(p)$
1	$Result_p \leftarrow \{o \mid l \in L \wedge l_s = p \wedge l_t = o\}$
2	$P_{invokedByObject} \leftarrow \{p_i \mid p_i \in P \wedge o \in Result_p \wedge 'e' \in l_{mod}(o, p_i)\}$
3	$P_{invokedDirect} \leftarrow \{p_j \mid p_j \in P \wedge l \in L \wedge l_s = p \wedge l_t = p_j \wedge' e' \in l_{mod}(p, p_j)\}$
4	Return $\{p \in P_{invokedByObject} \cup P_{invokedDirect} \wedge isReady(p)\}$

Algorithm 6 – findInvoked function

Using the $findInvoked$ function, we define the $hasInvoked(p)$ function, which yields true when the execution of a process triggered an event either by changing an object or by a direct invocation:

$$hasInvoked(p) = \begin{cases} true & findInvoked(p) \neq \emptyset \\ false & otherwise \end{cases}$$

Normally, the interpreter is in the $TopDown$ mode, in which it uses the global PC. When execution of a process is triggered by an event, the interpreter switches to $Event$ mode, where the use of the PC changes. Because of this, two changes related to event links are needed before updating the execution algorithm:

1. An additional version of $nextPC$ is provided, which receives an argument to be used as the current location of the PC instead of the global PC. This $nextPC$ version is used when execution returns to $TopDown$ mode. As this change is simple, it will not be further elaborated.
2. During the calculation of the dependencies of a process, agent event links are ignored by the algorithm. The reason for this is that the primary goal of agent event links is to control execution of processes, so counting them as dependencies complicates event-driven programs and can be a source of errors. This change is reflected in the new version of the $dep(p)$ function, shown in Algorithm 7.

	$dep(p)$
1	$L_{dep} = \{l \in L \wedge (l_{kind} \neq Agent \vee (l_{kind} = Agent \wedge' e' \notin l_{mod}))\}$
2	$D_{obj} = \{o \in (O_{par} \cup O_{var}) \wedge l \in L_{dep} \wedge l_s = o \wedge l_t = p\}$
3	$D_{state} = \{s \in S \wedge l \in L_{dep} \wedge l_s = s \wedge l_t = p\}$
3	Return $D_{obj} \cup D_{state}$

Algorithm 7 – Dependencies of process p excluding agent links with event control modifier

The new and final version of the OPP execution algorithm, shown in Algorithm 8, also stores $Mode$, the execution mode that is currently being performed, which is either $TopDown$ or $Event$.

	OPP Execution v4	
1	$PC \leftarrow \min(\{top(p) \mid p \in P\}), P_{wait} \leftarrow \{p \in P \wedge top(p) = PC\}$	
2	$P_{exec} \leftarrow \{p \in P_{wait} \wedge isReady(p)\},\ Mode \leftarrow TopDown$	
3	$P_{wait} = P_{wait} \setminus P_{exec}$	
4	While $P_{exec} \neq \emptyset$	
5	Wait for $p \in P_{exec}$ to finish	
6	$P_{exec} \leftarrow P_{exec} \setminus \{p\}$	
7	If $hasInvoked(p)$ Then $Mode \leftarrow Event, p_{inv} \leftarrow p$	
8	If $Mode = TopDown$ Then	
9	$PC_{cand} = nextPC$	TopDown mode
10	If $canAdvance(PC_{cand})$	
11	$executeNextProcesses$	
12	Else	
13	$executeCurrentProcesses$	
14	End If	
15	Else	
16	If $hasInvoked(p)$ then	Event mode
17	$P_{exec} \leftarrow P_{exec} \cup findInvoked(p)$	
18	Else If $P_{exec} = \emptyset$ then	
19	$Mode \leftarrow TopDown, PC \leftarrow bottom(p_{inv})$	
20	$executeNextProcesses$	
21	End If	
22	End If	
23	End While	
24	Error if $P_{waiting} \neq \emptyset$	

Algorithm 8 – In-Zoomed process execution with event links

We demonstrate the execution of this algorithm on the In-Zoomed OPD shown in Figure 47. As we have already worked out an example of top-down execution, we focus on the event-driven parts.

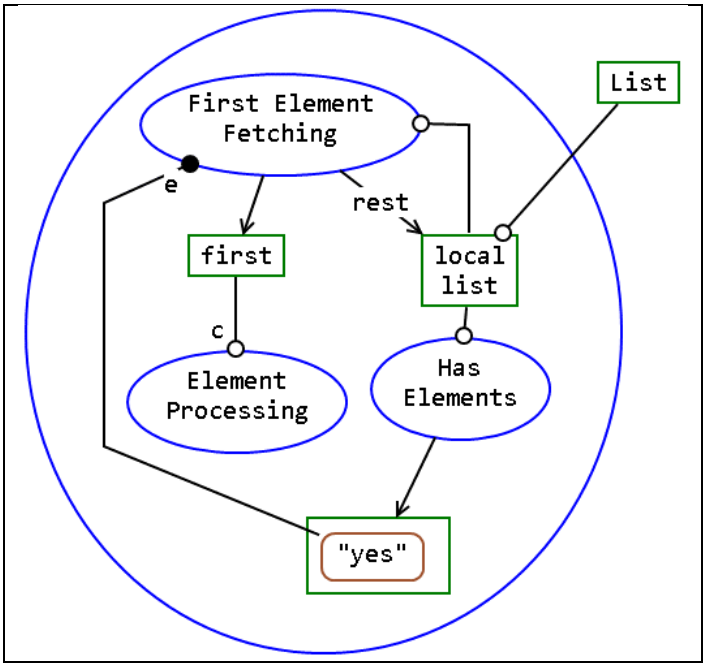


Figure 47 – Example In-Zoomed OPD with event links

The first process to be executed is **First Element Fetching**, which has only one dependency – **local list**. Note that the second incoming link to the process is an agent event link, so it does not count as a dependency. Assume that **local list** is a list that has at least two elements, and that process **Has Elements** finishes executing before **Element Processing** (line 6). Since **local list** has elements, the result of **Has Elements** is **yes**, so $Mode \leftarrow Event$ (line 7). This means that execution continues in line 15, and because a new process, **First Element Fetching**, was invoked, it is added to the executing set (lines 16-17), finishing the iteration.

After the next process, which can be either **First Element Fetching** or **Element Processing**, finishes executing, it is removed from the execution set. We know that no processes are invoked, the execution mode is $Event$, and there is one process in P_{exec}, so the interpreter starts a new iteration, waiting for the next process to finish (either **First Element Fetching** or **Element Processing**, depending which one finished in the previous iteration). After this process finishes $P_{exec} = \emptyset$, entering the condition in lines 18-21: The PC is set to the process that immediately follows the invoked process, the execution mode return to $TopDown$, and a new set of processes is executed, based on the top-down execution model, so **Element Processing** and **Has Elements** are executed again.

4 THE OPP RUNTIME ENVIRONMENT

As part of this work, we have also defined a runtime environment for the OPP language. This environment consists of an interpreter [59] of OPP programs and a number of built-in processes and predefined objects.

4.1 BUILT-IN TYPES AND COMPLEX TYPES

OPP has two simple built-in types: Number and String. OPP also has two built-in collection types: List and Set. More types, called "Complex Types", can be defined using Type OPDs. Furthermore, OPP has a type called "Any", which can be used to store any type of object (simple, collection, and complex). This hierarchy is shown in Figure 48 using Generalization-Classification.

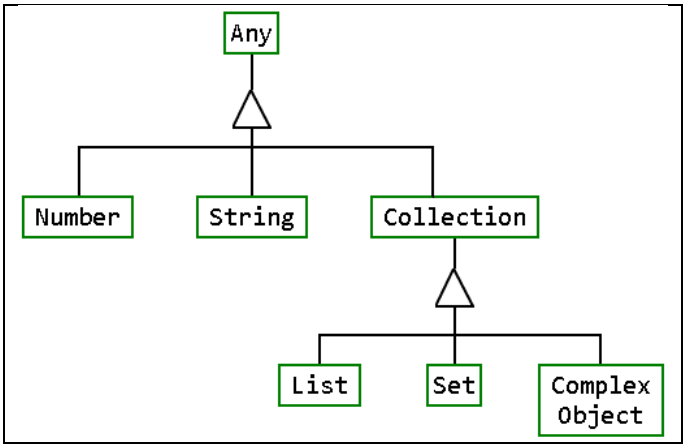


Figure 48 – Built-in types object hierarchy

A Set is an unordered collection of elements with no duplicates ids, as defined in Section 3.6. A List is an ordered collection of indexed elements. The first index of the list is 1 and the last index equals the number of element in (or size of) the list. List contents are managed using built-in processes, which are described below.

Objects in OPP can be created by invoking the "Create Object" process described in Section 4.3.1, or by initializing them in-place using JSON [60] notation. For simple types, this is done by adding the value to the object in its definition, as shown in Figure 49.

`a: Number = 5`	`b: String = "Hello World"`
Number initialization	String Initialization

Figure 49 – Simple type initialization

Collections are initialized using JSON arrays. If the type of the object is not given, the runtime defaults to using a List, as exemplified in Figure 50. If the object being initialized is

typed as a Set, the interpreter will check in runtime that the initializer does not contain duplicate elements, and it will issue an error message in case this happens.

```
c = ["a",1,"c",3,5]
```

Figure 50 – Collection initialization

Complex types are initialized using JSON objects. If the type of the object is given, the interpreter will validate in runtime that the target object has all the parts defined in the JSON object, and it will issue an error message if this is not the case. This check is done deeply, so if a complex object has parts which are themselves complex objects, their parts will also be validated. If the JSON object does not contain a part that is defined in the complex object, this part will be left uninitialized. An example of complex object initialization is shown in Figure 51.

```
d: Cart = {owner:"user1",items:["banana","mango"],price:35}
```

Figure 51 – Complex object initialization using JSON

Another way to initialize complex objects is by giving a value to its parts in the OPD in which they are defined. An example of doing this for the object shown above is shown in Figure 52.

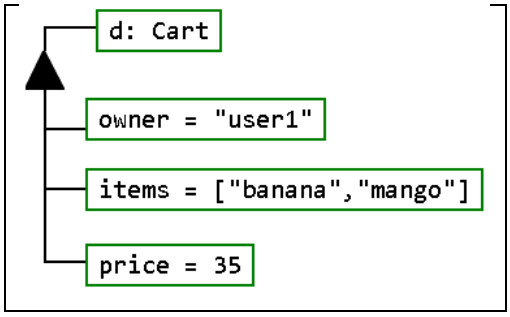


Figure 52 – Complex object initialization using part initialization

4.2 CONTEXT

An executing process is always provided with a *context* parameter object, which contains the following data:

- **Process Name** – the name of the process being executed,
- **Arguments** – the arguments that were given to the process, provided as a complex object where each part is one argument,
- **Runtime Data** – which can be used by the interpreter at runtime, the pre-executing processes, the process itself, or post-executing processes to transfer data between them.

The visual representation of the **Context** object is shown in Figure 53.

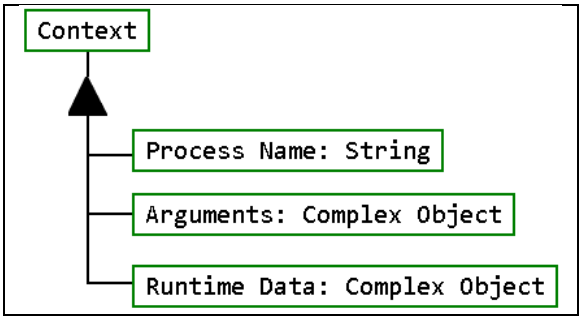


Figure 53 - Context object

A good example of using the **Context** object is the case of processes that require authentication and authorization, presented in Section 3.6. Authentication and authorization are required by many Web-based services, yet adding them to the business logic of a process creates much duplication and complicates the process with non-business-related logic. To solve this, authentication and authorization can be done in OPP by a pre-executing process of an abstract process, and then all processes that require authentication and authorization can inherit from this process. This approach is similar to that taken by aspect-oriented programming [61].

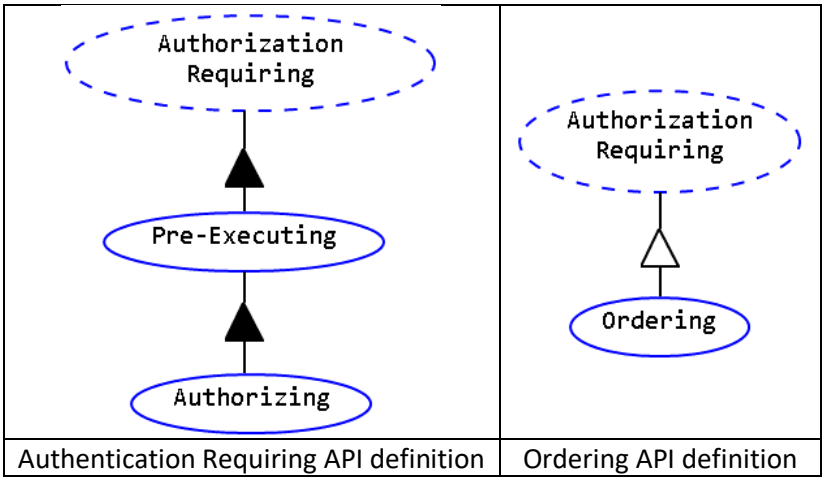


Figure 54 – Definition of Authentication Requiring and Ordering processes

As shown in Figure 54, we define an abstract process called **Authorization Requiring**, whose only goal is to add the pre-executing **Authorizing** process. Then we defined the process **Ordering**, which classifies **Authorization Requiring**, implying that before the process **Ordering** is executed, the process **Authorizing** is invoked. The In-Zoomed OPD for **Authorizing** is shown in Figure 55.

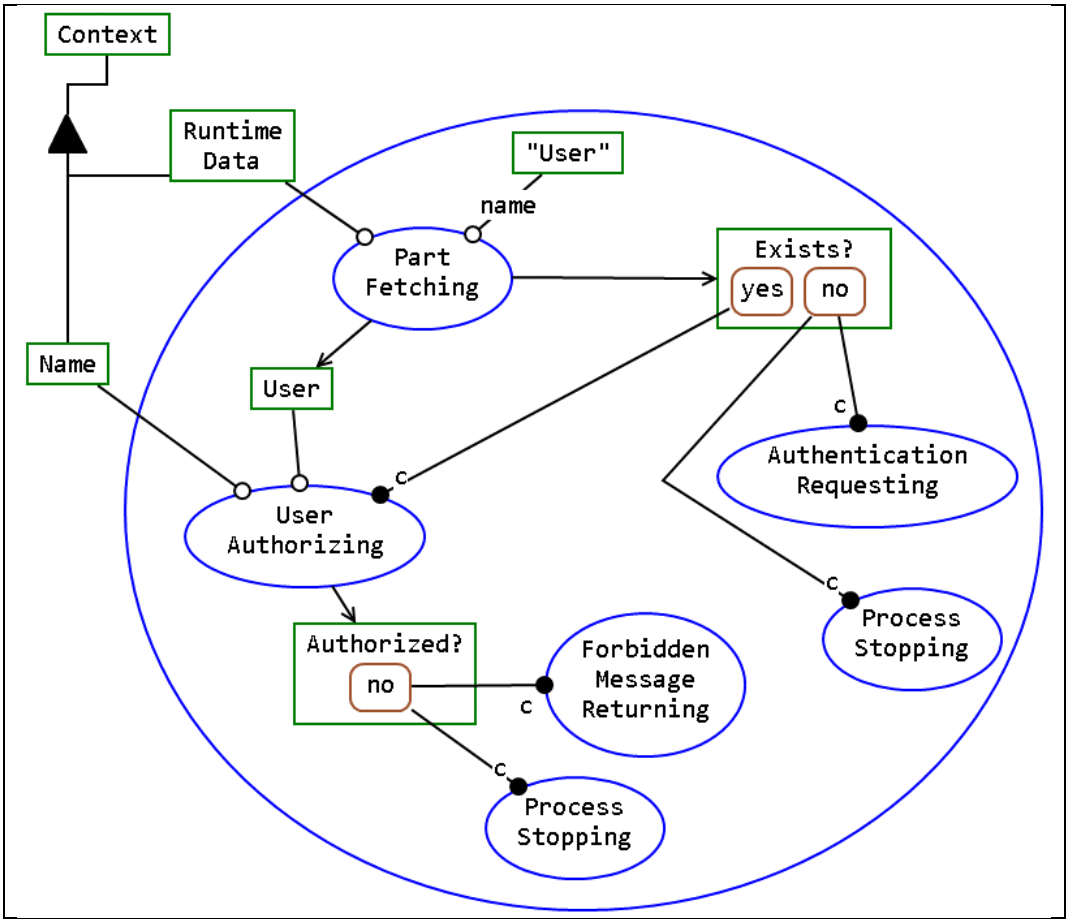


Figure 55 – Authorizing In-Zoomed

First, we check if the runtime contains a user, by fetching the **User** part from the runtime data. If it does, we validate if **User** is authorized to perform this operation using the process **User Authorizing**. If **Part Fetching** yields that **Exists?** is in state **no** (meaning that user does not exist in the runtime data), the user has not performed authentication, so the process **Authentication Requesting** is invoked, and then **Process Stopping** is executed, which stops execution of **Ordering** and all pre-executing processes, returning control to the calling process. If **Exists?** is in state **yes** (so the user is already authenticated), the process **User Authorizing** is executed, giving it the **Name** of the process. If **Authorized?** is yielded in state **no** (meaning that the user is not authorized to perform this process), **Forbidden Message Returning** and **Process Stopping** are executed. If the user is both authenticated and authorized, **Authorizing** finishes, and after all the other pre-executing processes are executed, **Ordering** is executed. Some of the processes used by **Authorizing** are built-in and will be explained later in this section. Other processes are domain-specific, and it is assumed that they have been implemented by the developer of the program.

4.3 Built-In Processes

The OPP interpreter comes with some built-in processes commonly required in software systems. A built-in process can have multiple aliases, which are multiple names that refer to the same process.

4.3.1 General

1. **Object Creating**
    a. **Description**: Create a new object of Type **type**. If no type is given, the **object** is of **Complex Object** type. **String** and **Number** objects cannot be created using this process as they must have a value assigned to them at creation time.
    b. **Aliases**: **Create Object**
    c. **Interface OPD**:

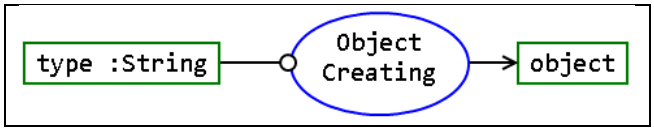


Figure 56 – Object Creating interface OPD

    d. **Example**:

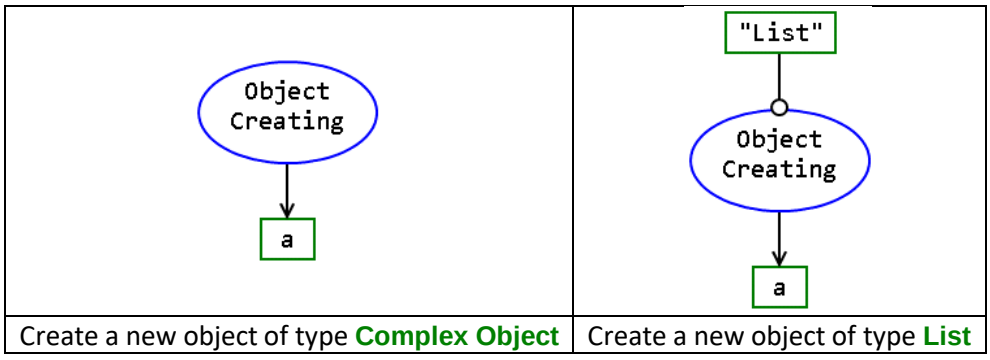


Figure 57 – Object Creating example

2. **Process Copying**
    a. **Description**: Create a **copy** object, giving it the same value as **object**, including all parts, recursively
    b. **Aliases**: **Copy**
    c. **Interface OPD**:

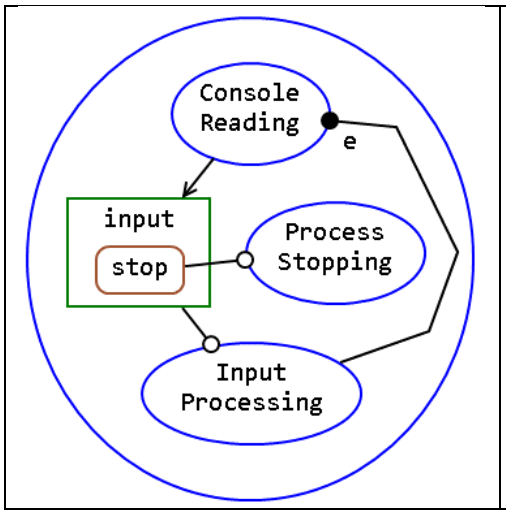
Figure 58 – Object Copying interface OPD

3. **Process Stopping**

a. **Description**: Stop the execution the current process, returning control to the calling process or the interpreter.
b. **Aliases**: **Stop Process**
c. **Interface OPD**:

Figure 59 – Process Stopping interface OPD

d. **Example**:



The process is in a loop that reads input from the console and processes the input. If the user enters "stop" in the console, **Process Stopping** is invoked and the loop is exited

Figure 60 – Process Stopping example

4.3.2 Math

1. **Adding**
   a. **Description**: Add parameters **a** and **b** and yield the result in **c**.
   b. **Aliases**: **+**
   c. **Interface OPD**:

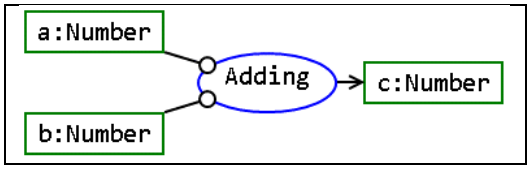


Figure 61 – Adding interface OPD

d. **Example**: the value of **c** will be 3 after executing the process.

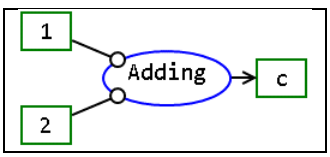


Figure 62 – Adding example

2. **Subtracting**
    a. **Description**: Subtract parameter **b** from parameter **a** and yield the result in **c**.
    b. **Aliases**: **-**
    c. **Interface OPD**:

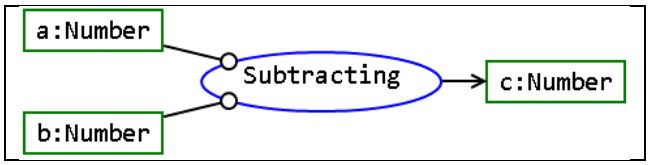


Figure 63 – Subtracting interface OPD

    d. **Example**: the value of **c** will be "2" after executing the process. Since subtraction is not commutative, the arguments given to the process must be named parameters (or any other parameter matching method, as described in Section 3.7.2.2).

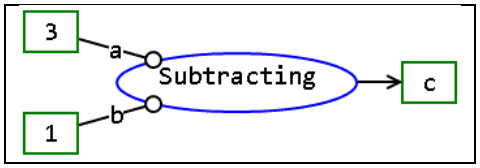


Figure 64 – Subtracting example

3. **Multiplying**
    a. **Description**: Multiply parameter **a** and **b** and yield the result in **c**.
    b. **Aliases**: *
    c. **Interface OPD**:

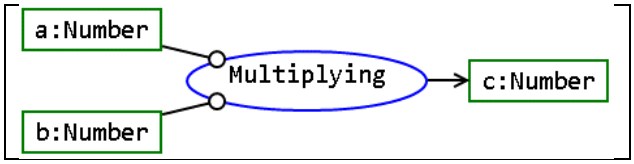


Figure 65 – Multiplying interface OPD

    d. **Example**: the value of **c** will be 15 after executing the process.

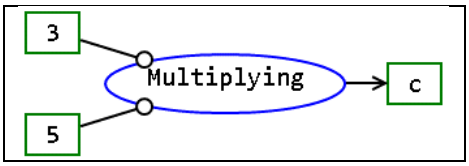


Figure 66 – Multiplying example

4. **Dividing**
    a. **Description**: Divide parameter **a** by parameter **b** and yield the result in **c**.

b. **Aliases**: **/**

c. **Interface OPD**:

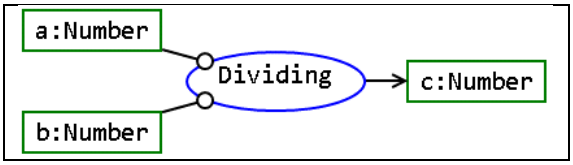


Figure 67 – Dividing interface OPD

d. **Example**: the value of **c** will be 7 after executing the process. Similar to subtraction, argument matching is important here, as division is not commutative

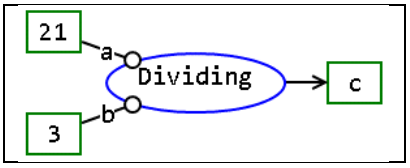


Figure 68 – Division example

5. **Number Comparing**

a. **Description**: Compare two numbers **a** and **b**, yielding in **c** the value **-1**, **0** or **1** if **a** is smaller, equals, or greater than **b**, respectively

b. **Aliases**: **Number Compare**

c. **Interface OPD**:

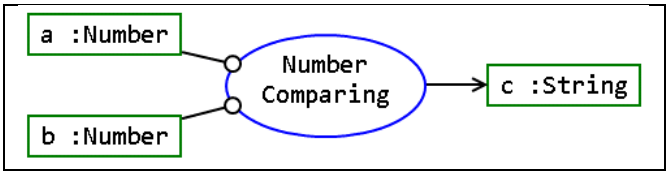


Figure 69 – Number Comparing interface OPD

d. **Example**: the value of **c** will be **-1** after executing the process

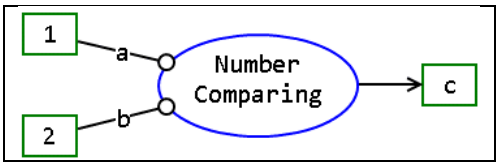


Figure 70 – Number Comparing example

4.3.3 Strings

1. **Concatenating**

a. **Description**: Concatenate two **String** objects.

b. **Aliases**: **concatenate**

c. **Interface OPD**:

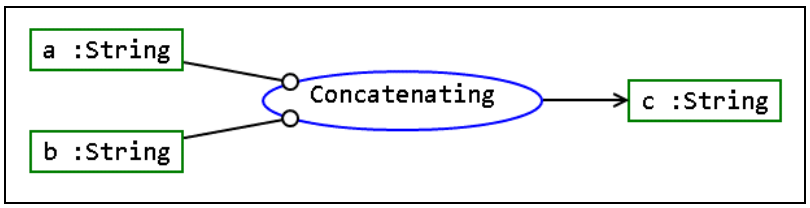


Figure 71 – Concatenating interface OPD

d. **Example**:

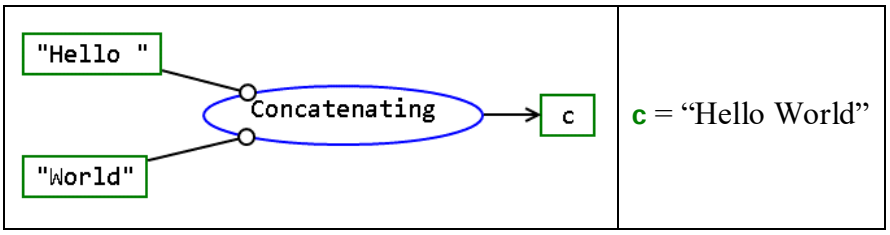


Figure 72 – Concatenating example

2. **String Comparing**
   a. **Description**: Compare two strings **a** and **b**, yielding in **c** the value **-1**, **0**, **1** if a lexicographically precedes, equals, or succeeds **b**
   b. **Aliases**: **Compare Strings**
   c. **Interface OPD**:

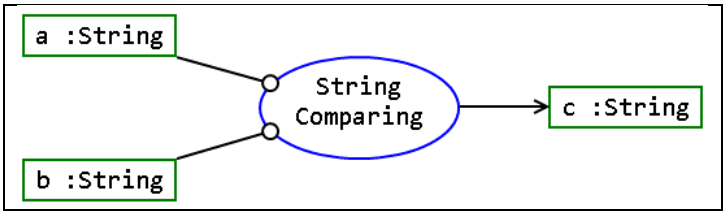


Figure 73 – String Comparing interface OPD

   d. **Example**: the value of **c** will be **-1** as **"Hello"** is before **"World"** lexicographically

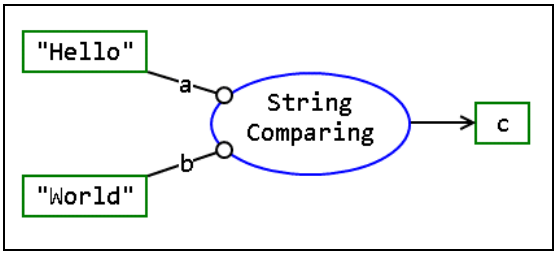


Figure 74 – String Comparing example

4.3.4 Collections and Complex Objects

For all examples in this section, we will use the collections specified in Figure 75.

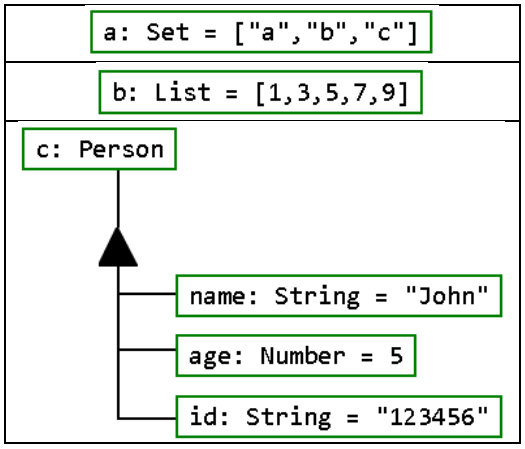


Figure 75 – Example collections for process definition examples

4.3.4.1 General

1. **Element Counting**
   a. **Description**: return the number of elements in the collection (the collection size).
   b. **Aliases**: **Count**
   c. **Interface OPD**:

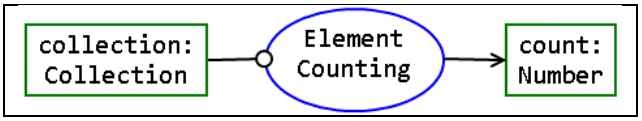


Figure 76 – Element Counting interface OPD

   d. **Example**:

Collection Type	Example	Yielded Values
Set	a → Element Counting → count	**count** = 3
List	b → Element Counting → count	**count** = 5

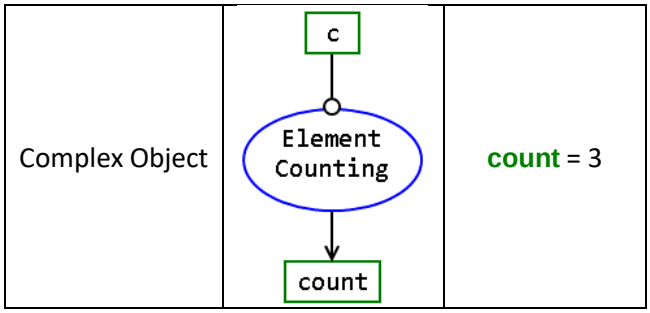


Figure 77 – Element Counting example

4.3.4.2 List

1. **First Element Adding**
   a. **Description**: Add an element to the start of the list.
   b. **Aliases**: **Add First**
   c. **Interface OPD**:

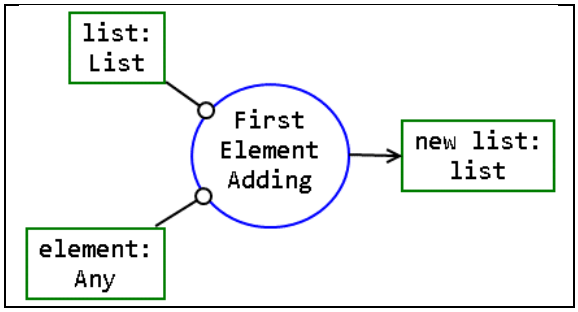


Figure 78 – First Element Adding interface OPD

   d. **Example**:

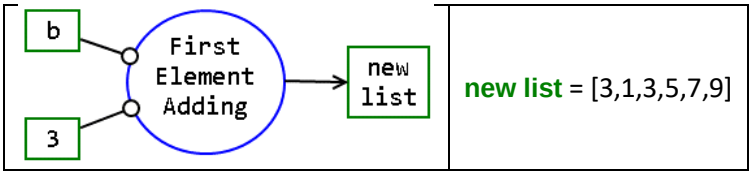


Figure 79 – First Element Adding example

2. **First Element Fetching**
   a. **Description**: Fetch the first element of the list. If the list is empty, no element is returned. The object **fetched?** is "yes" if an element was fetched and "no" if the list is empty.
   b. **Aliases**: **Get First**
   c. **Interface OPD**:

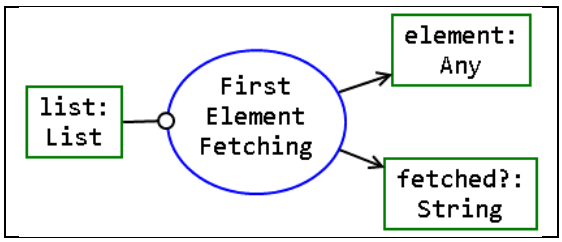


Figure 80 – First Element Fetching interface OPD

d. **Example**:

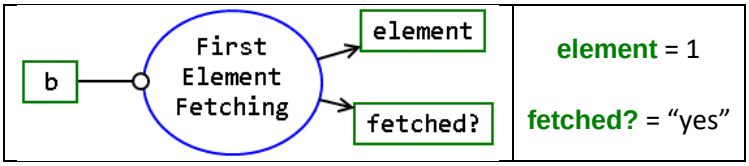


Figure 81 – First Element Fetching example

3. **First Element Removing**
   a. **Description**: Remove the first element of the list, returning the element, a list without the removed element, and a flag object to check if an element was removed. If the list is empty, **element** will be empty, **new list** will be an empty list, and **fetched?** will be "no". In all other cases, **fetched?** will be "yes".
   b. **Aliases**: **Remove First**
   c. **Interface OPD**:

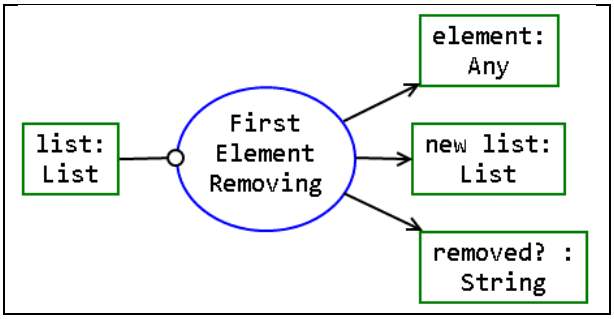


Figure 82 – First Element Removing interface OPD

   d. **Example**:

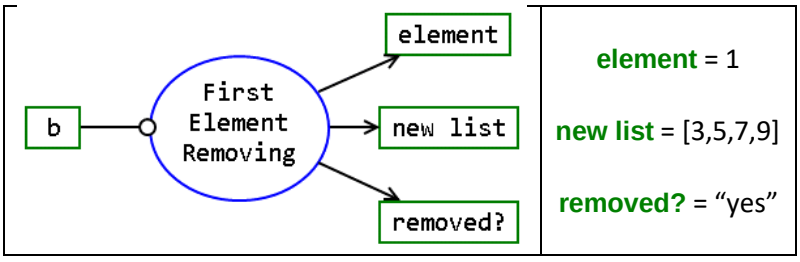


Figure 83 – First Element Removing example

4. **Location Element Adding**

a. **Description**: Add an element to a list at a specific location. Lists have a starting index of 1 and an ending index of the number of elements (the size) of the list. If **location** is outside these indexes, the element in not added, **added** will be "no" and **new list** will be empty. Otherwise **new list** will contain a list where **element** is located at **location** and all elements in **list** after this location will have their location increased by 1.

b. **Aliases**: **Add Element**

c. **Interface OPD**:

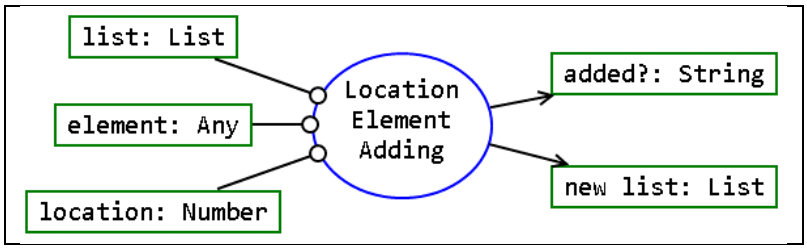


Figure 84 – Location Element Adding interface OPD

d. **Example**:

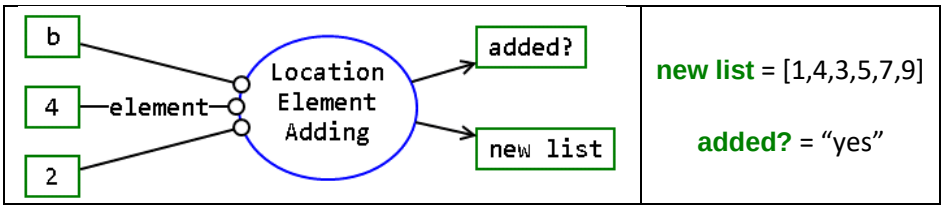


Figure 85 – Location Element Adding example

5. **Location Element Fetching**

a. **Description**: Fetch and element at a specific location. If the location is outside the bound of the list, **fetched?** will be "not" and **element** will be empty.

b. **Aliases**: **Fetch Element**

c. **Interface OPD**:

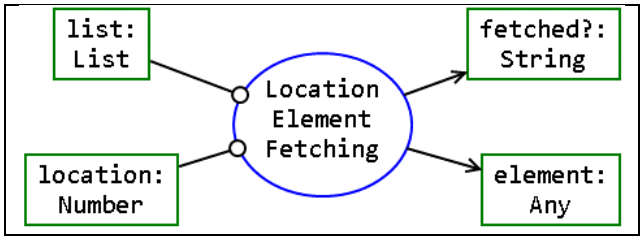


Figure 86 – Location Element Fetching interface OPD

d. **Example**:

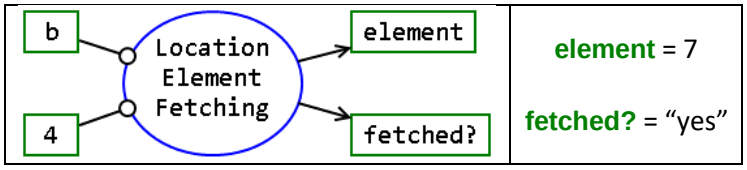


Figure 87 – Location Element Fetching example

6. **Location Element Removing**
   a. **Description**: Remove an element from a specific location in the list. After removing the element, all other elements in the list are shifted left so their location is now one less than the location before the element was removed. If the location is outside the bounds of the list, **element** will be empty, **new list** will be empty, and **removed?** will be "no". In all other cases, **element** will contain the object removed from **location**, and **removed?** will be "yes".
   b. **Aliases**: **Remove Element**
   c. **Interface OPD**:

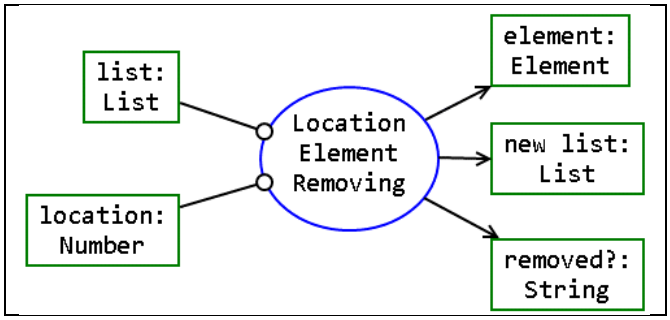


Figure 88 – Location Element Removing interface OPD

   d. **Example**:

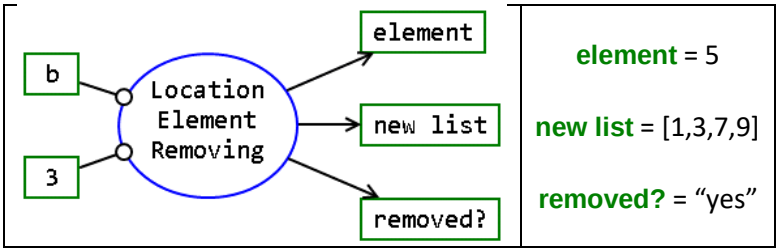


Figure 89 – Location Element Removing example

7. **Last Element Adding**
   a. **Description**: Add an element at the end of the list.
   b. **Aliases**: **Add Last**
   c. **Interface OPD**:

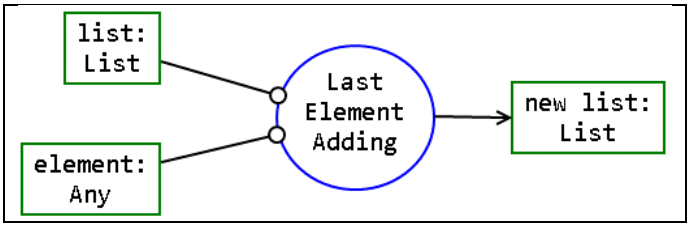


Figure 90 – Last Element Adding interface OPD

d. **Example**:

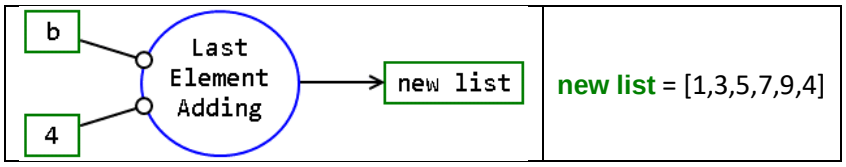


Figure 91 – Last Element Adding example

8. **Last Element Fetching**
   a. **Description**: Fetch the last element of the list. If **list** is empty, **element** will be empty. **fetched?** is "yes" if an element was fetched, and "no" if **list** is empty.
   b. **Aliases**: **Get Last**
   c. **Interface OPD**:

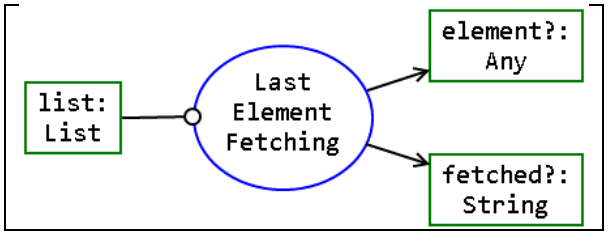


Figure 92 – Last Element Fetching interface OPD

   d. **Example**:

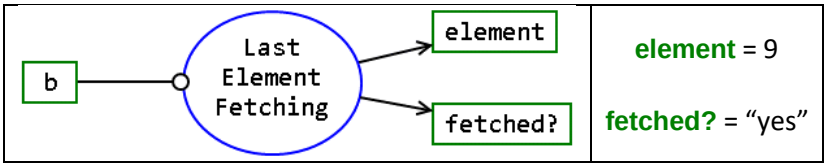


Figure 93 – Last Element Fetching example

9. **Last Element Removing**
   a. **Description**: Remove the last element of the list, returning the element, a list without the removed element, and a flag object indicating if an element was removed. If **list** is empty, **element** will be empty, **new list** will be an empty list, and **fetched?** will be "no". In all other cases, **fetched?** will be "yes".
   b. **Aliases**: **Remove Last**
   c. **Interface OPD**:

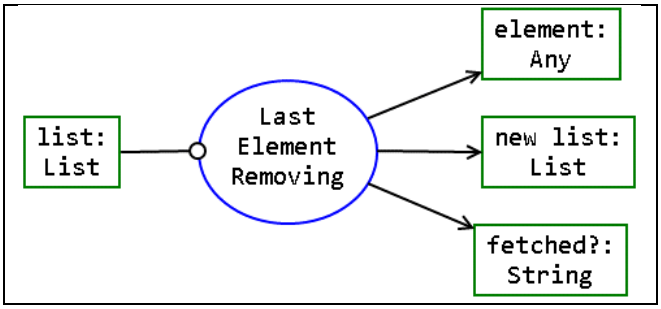


Figure 94 – Last Element Removing interface OPD

d. **Example**:

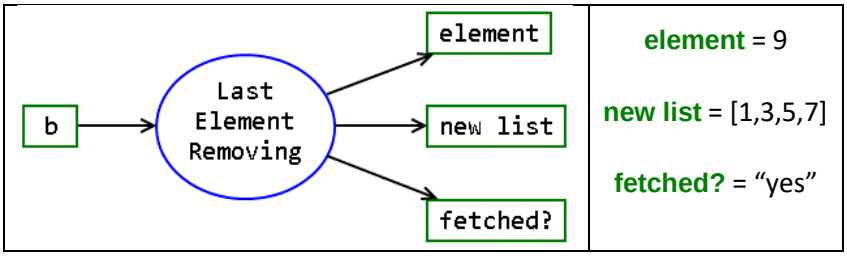


Figure 95 – Last Element Removing example

4.3.4.3 Complex Object

1. **Part Adding**
   a. **Description**: Add a **part** to a complex object using the Aggregation-Participation relation. If the object already contains a part named **name**, that part is replaced. The process yields **new object**, which contains the added **part**, **replaced?** which is "yes" if the part was replaced, and "no" otherwise, and **replaced part**, which contains the value of the replaced part in case **replaced?** is "yes".
   b. **Aliases**: **Add Part**
   c. **Interface OPD**:

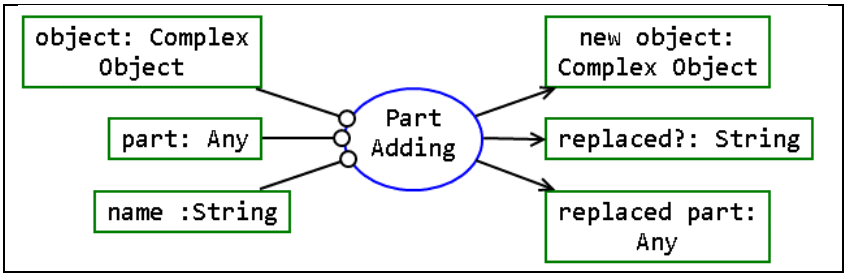


Figure 96 – Part Adding interface OPD

d. **Example**:

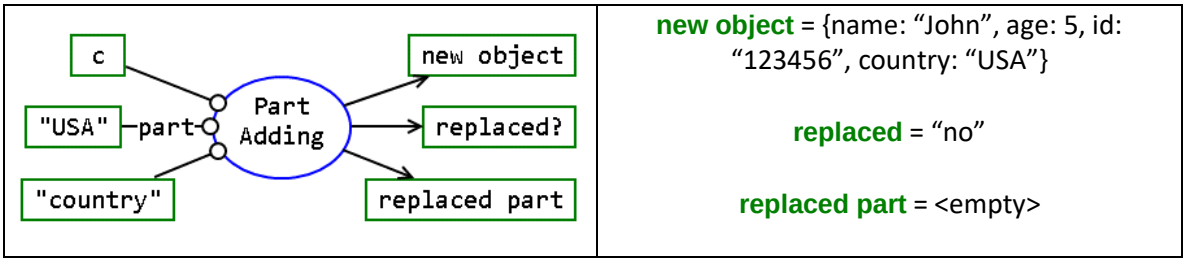


Figure 97 – Part Adding example

2. **Part Fetching**
   a. **Description**: Fetch a part from a complex object. The process yields **fetched?** which is “yes” if the object has a part named **name** and “no” if there is no part with that name, and **part**, which contains the value of the fetched part.
   b. **Aliases**: **Fetch Part**
   c. **Interface OPD**:

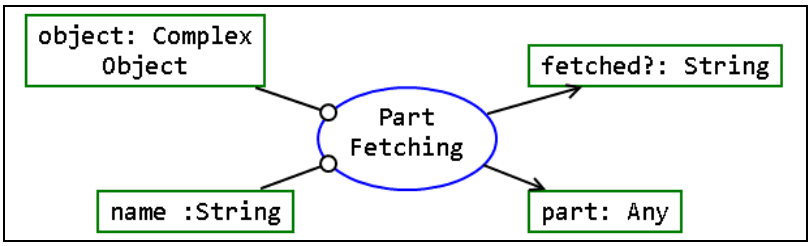


Figure 98 – Part Fetching interface OPD

   d. **Example**:

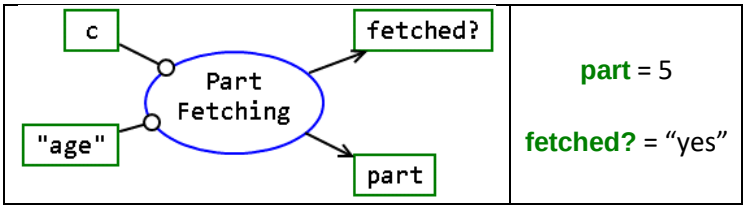


Figure 99 – Part Fetching example

3. **Part Removing**
   a. **Description**: Create a new object with the given part removed. If there is no part named **name**, **new object** is a copy of **object**, and **removed?** is “no”. Otherwise **removed?** is “yes”, and **part** contains the value of the removed part.
   b. **Aliases**: **Remove Part**
   c. **Interface OPD**:

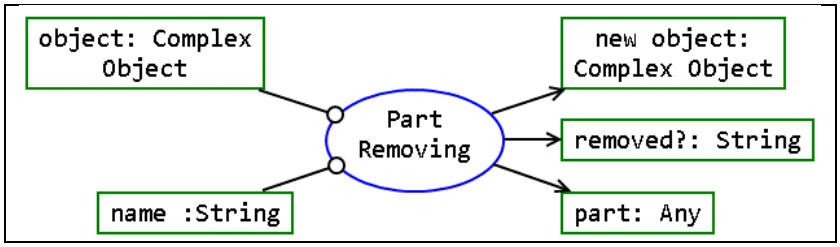


Figure 100 – Part Removing interface OPD

d. **Example**:

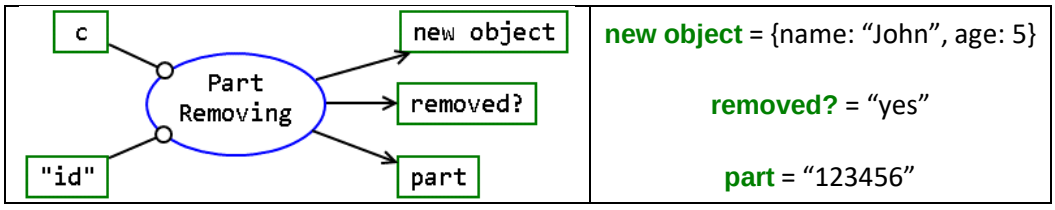


Figure 101 – Part Removing example

4. **All Parts Fetching**
   a. **Description**: Fetch all the parts of **object** as a **List**.
   b. **Aliases**: **Fetch Parts**
   c. **Interface OPD**:

Figure 102 – All Parts Fetching interface OPD

d. **Example**:

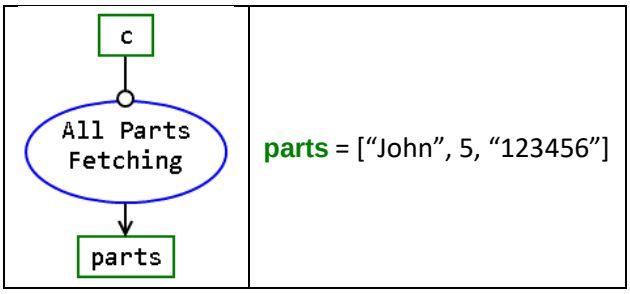


Figure 103 – All Parts Fetching example

5. **All Part Names Fetching**
   e. **Description**: Fetch the name of all parts of **object** as a **List**.
   f. **Aliases**: **Fetch Part Names**
   g. **Interface OPD**:

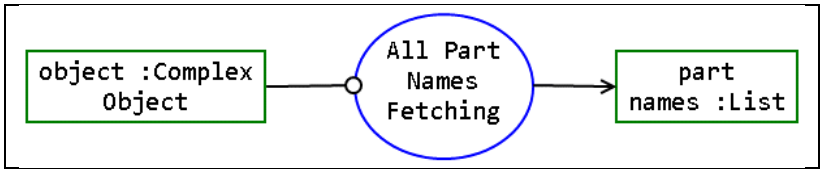


Figure 104 – All Part Names Fetching interface OPD

h. **Example**:

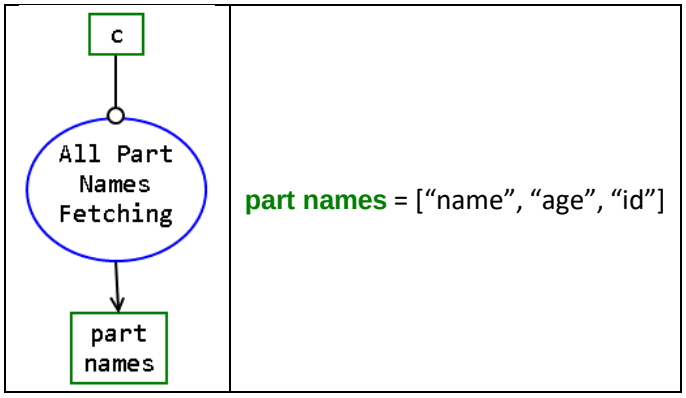


Figure 105 – All Part Names Fetching example

4.3.5 Input and Output

These processes are used to get input from the user or give output to the user. In all the following processes, when the input is treated as a JSON string and the JSON string contains an array, the object that is created by these processes is of type **List**.

1. **Console Reading**
   a. **Description**: Read an object from the console. The user is shown **prompt** and the process waits for input. If the input from the console can be treated as a number, an object of type **Number** is created. If the input can be treated as a string, an object of type **String** is created. Otherwise, the input is read as a JSON formatted string, and a **Collection** is created. If the input can be parsed, **parse error?** is "no". If the input cannot be parsed by any method, no object is returned and the **parse error?** object will be "yes".
   b. **Aliases**: **Console Input**
   c. **Interface OPD**:

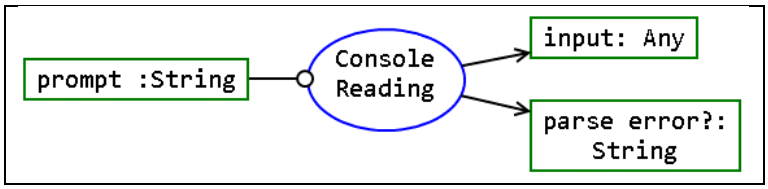


Figure 106 – Console Reading interface OPD

d. **Example**: The user is shown the text “Please enter a number”. Assuming that the user entered “5.4646” on the console, the result is presented in Figure 107.

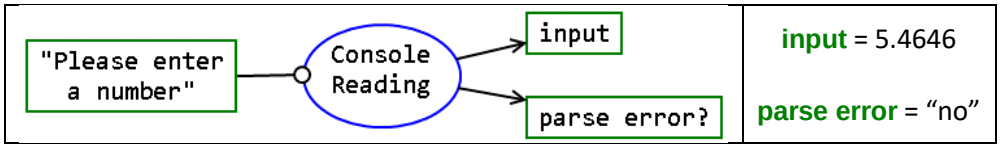


Figure 107 – Console Reading example

2. **Console Writing**

a. **Description**: Write an object to the console. Simple objects (**String**, **Number**) are written with no formatting. Collections are written using JSON notation.

b. **Aliases**: **Console Output**

c. **Interface OPD**:

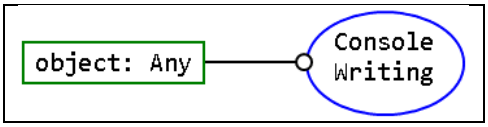


Figure 108 – Console Writing interface OPD

d. Example:

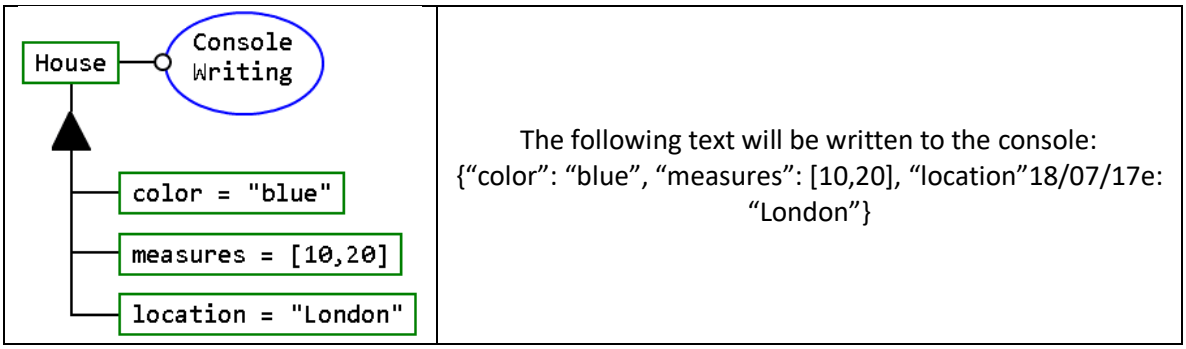


Figure 109 – Console Writing example

3. **Dialog Text Reading**

a. **Description**: Read an object from a dialog shown to the user. The user is shown **prompt** and the process waits for input. If the input can be treated as a number, an object of type **Number** is created. If the input can be treated as a string, an object of type **String** is created. Otherwise the input is read as a JSON formatted string and a **Collection** is created. If the input can be parsed, **parse error?** is “no”. If the input cannot be parsed by any method, no object is returned and **parse error?** will be “yes”.

b. **Aliases**: **Dialog Input**

c. **Interface OPD**:

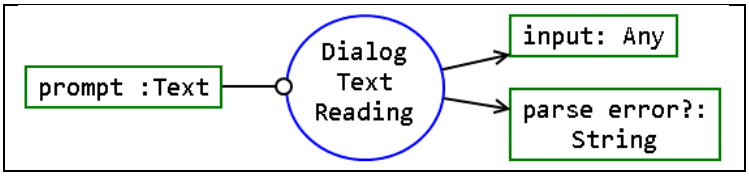


Figure 110 – Dialog Text Reading interface OPD

d. **Example**: The user is shown the text "Please enter a number". Assuming that the user entered "Hello World!" on the console, Figure 111 shows the result.

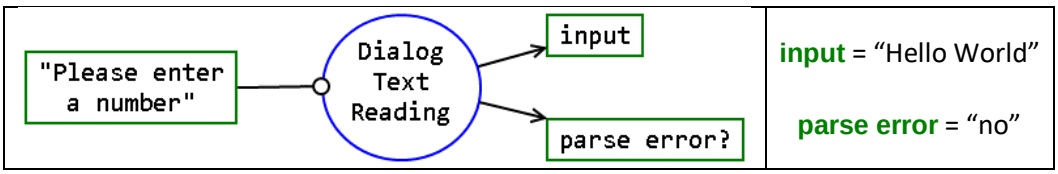


Figure 111 – Dialog Text Reading example

4. **Dialog Text Writing**
   a. **Description**: Write an object to a dialog shown to the user (this process is only available when the runtime is executed as a stand-alone program and not as a web service). Simple objects (**String**, **Number**) are written with no formatting. Collections are written using JSON notation.
   b. **Aliases**: **Dialog Output**
   c. **Interface OPD**:

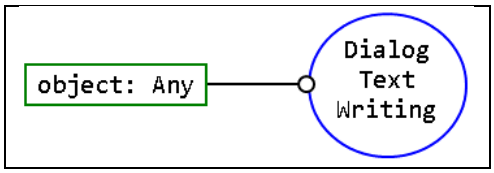


Figure 112 – Dialog Text Writing interface OPD

   d. **Example**:

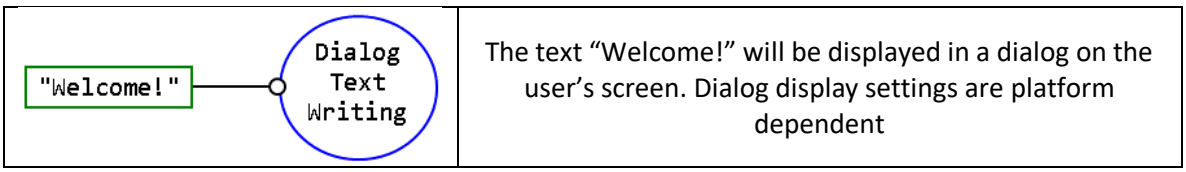


Figure 113 – Dialog Text Writing

5. **Text File Reading**
   a. **Description**: Read an object from a text file. The name of the file is given in **file name**. If the input can be treated as a number, an object of type **Number** is created. If the input can be treated as a String, an object of type String is created. Otherwise the input is read as a JSON formatted string and a Collection is created. If the input can be parsed, **parse error?** is "no". If the

input cannot be parsed by any method, no object is returned and **parse error?** will be "yes". If there was an error reading the file, **file error?** will be "yes", otherwise it will be "no".

b. **Aliases**: **Read Text File**

c. **Interface OPD**:

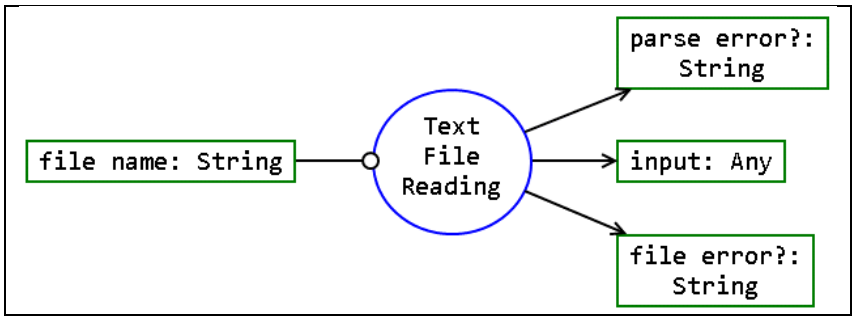


Figure 114 – Text File Reading interface OPD

d. **Example**: assuming the file "input.txt" exists in the working directory and contains the text "[4,5, "hello", 4]".

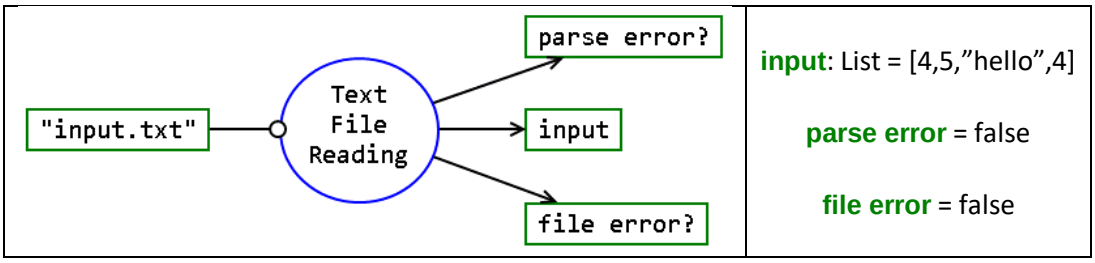


Figure 115 – Text File Reading example

6. **Text File writing**

a. **Description**: Write an object to a file in textual format. The name of the file is given in **file name**. Simple objects (**String**, **Number**) are written with no formatting. Collections are written using JSON notation. If there is an error writing to the file, **file error?** will be "yes", otherwise it will be "no".

b. **Aliases**: **Write Text File**

c. **Interface OPD**:

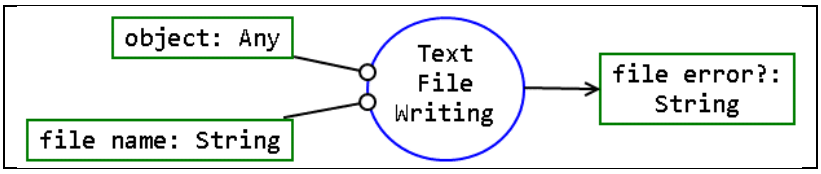


Figure 116 – Text File Writing interface OPD

d. **Example**:

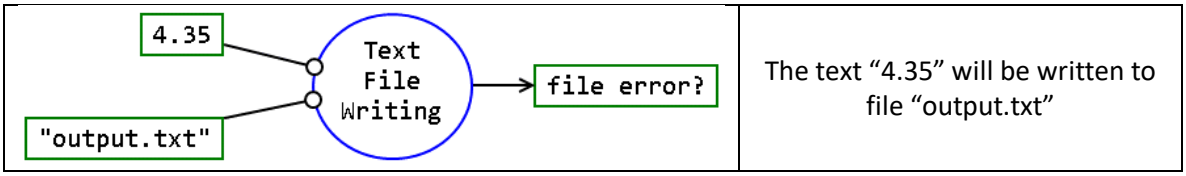


Figure 117 – Text File Writing example

5 THE OPP DEVELOPMENT ENVIRONMENT

The development environment where OPP programs are created and interpreted was developed as a plugin to the Eclipse [62] integrated development environment (IDE), using the EMF and GEF frameworks also under the umbrella of the Eclipse foundation. We decided to use this platform as it is considered one of the most popular development platforms, it is open source, and has advanced tools and frameworks for the development of graphical editors. Figure 118 shows a screenshot of the IDE displaying an In-Zoomed OPD.

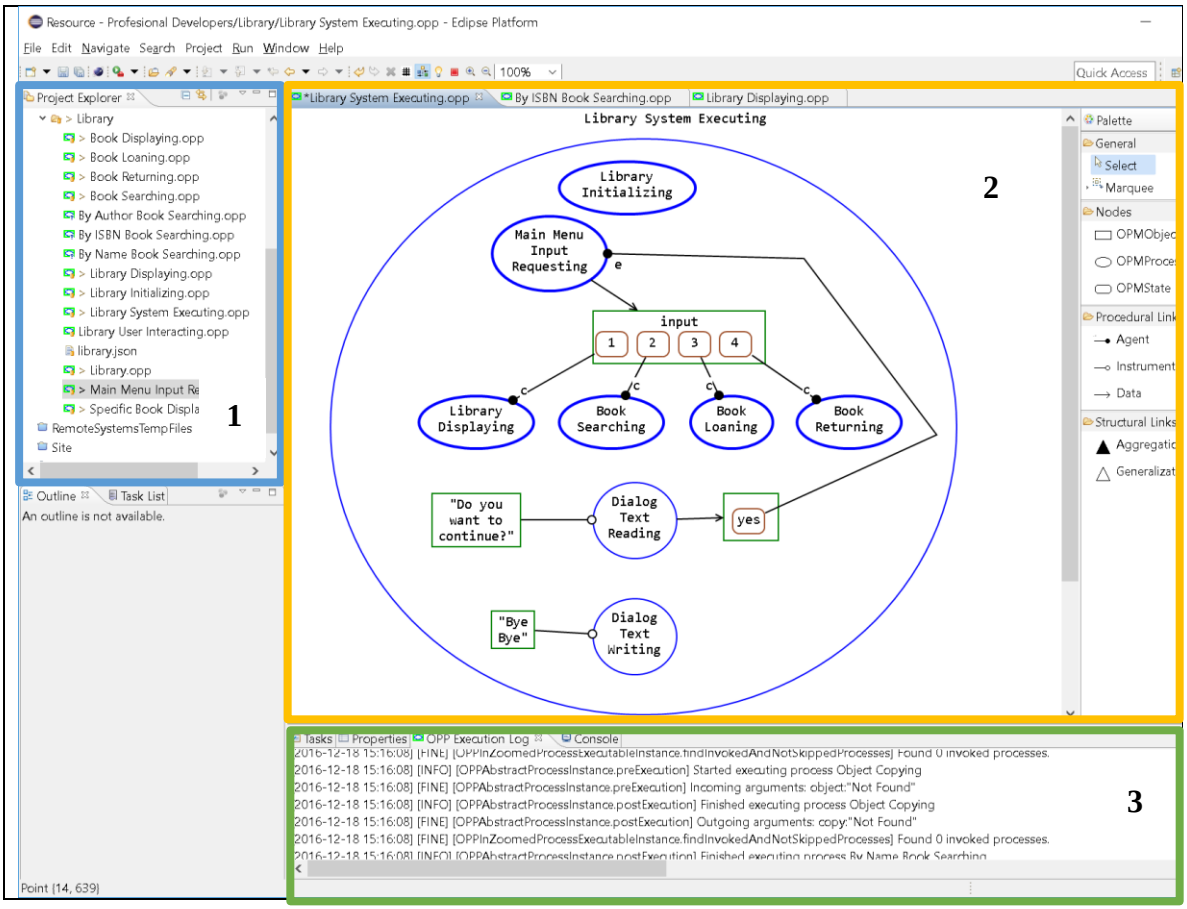


Figure 118 – The OPP IDE

The main parts of the development environment are the Project Explorer (1 – outlined in blue), the OPD Editor (2 – outlined in yellow), and the OPP Execution Log (3 – outlined in green).

1. Project Explorer: An OPP program/system is defined inside a project. When an In-Zoomed process is executed, the interpreter will search for processes inside a project. At this stage the interpreter supports only one hierarchy level for each project. This limitation will be removed in the future, allowing for some packaging/module management mechanism.
2. OPP Editor: This is where OPDs are created and modified. When an OPD is open, the name of the OPD is shown at the top of the editing window. In the current implementation of the OPP interpreter, the name of the OPD matches the name of the file in which the OPD is stored, but this is not a requirement of the language itself. The user can interact with the editor using the entities palette located at the right of the editor, and can also perform multiple editing operations using the mouse and the keyboard
3. OPP Execution Log: When an OPP program is executed, this window shows the order of execution of the processes in the program, what arguments are passed two and from invoked processes, and any errors that may occur during program execution

The OPP editor and interpreter both rely on an EMF model that stores each OPD as observable and serializable Java objects. The editor is implemented using the Model-View-Controller (MVC) [63] design pattern, which connects between the EMF model layer with the GEF controller and view layer. The interpreter is a standalone Java program that loads the system from OPP files created by the editor (and serialized to XML by the EMF framework) and interprets it.

The source code of the editor and the interpreter can be found at: http://github.com/vainolo/Object-Process-Programming

6 USE CASE – ABS SYSTEM

To show how OPP can be used to extend the conceptual modeling capabilities of OPM into fully executable models, the ABS – anti-lock braking system – OPM model example that comes with OPCAT is fully implemented in OPP.

ABS is a safety system used in automobiles, which prevents wheels from locking up while breaking and helps avoid car skidding. The system diagram of the ABS system as modeled in OPM is shown in Figure 119.

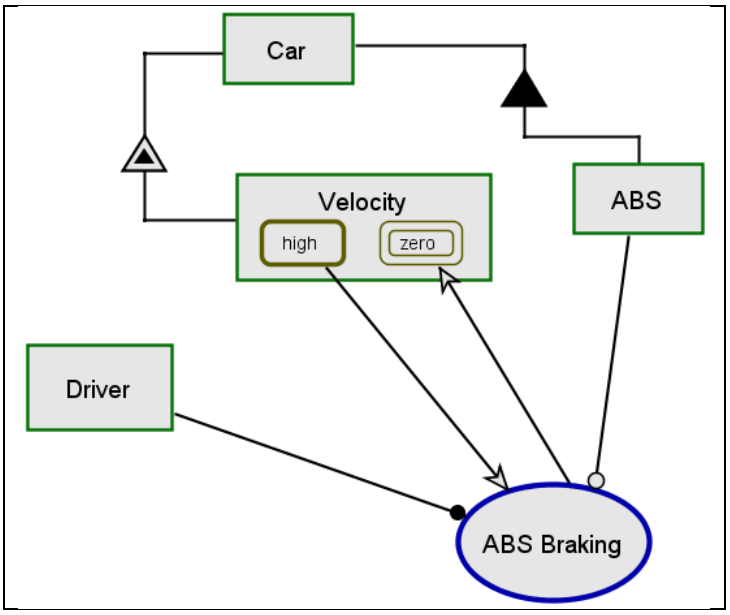


Figure 119 – ABS System Diagram in OPM

When development of the system moves from the conceptual phase to the implementation phase, the **ABS Breaking** process is created in OPP, as show in Figure 120.

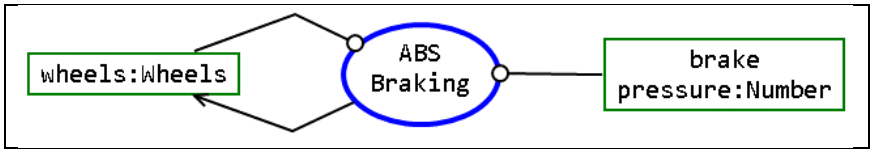


Figure 120 – ABS Process in OPP

Already at this stage, the difference between the conceptual model and the implementation are apparent. The conceptual model deals with how the system works from the perspective of the system's stakeholders, in this case the **Driver** and the **Car**. In contrast, the implementation looks at what data is required to perform the process, and what data is created by the process. The **Driver** in the conceptual model is replaced by the **brake pressure** parameter, as this is the way the driver indicates the need to brake. The **ABS** and **Velocity** of the **Car** are replaced by the **wheels**, which are the actual objects affected by the breaking, and from which all other required information can be fetched. The Wheels and Wheel types used

in the model are shown in Figure 121. As previously stated, types are not required by OPP, but they make the system easier to understand and reduce type errors throughout the program.

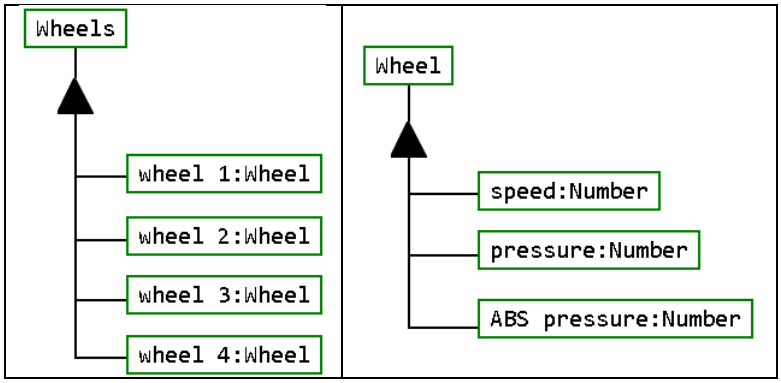


*Figure 121 – **Wheels** and **Wheel** OPP types*

The **ABS Breaking** progress works as follows: the desired pressure is stored in each wheel, and then the real pressure that must be applied to the wheels is calculated, based on the speed of the wheel compared to the speed of the other wheels. This calculated pressure is then set on each of the wheels, and to simulate the process, the velocity of the wheels is updated based on the applied pressure and the current velocity of the wheel. This process is shown in Figure 122. In the real end-system this last step would be removed as the wheels are constantly braking based on the pressure that is applied to them in the previous steps.

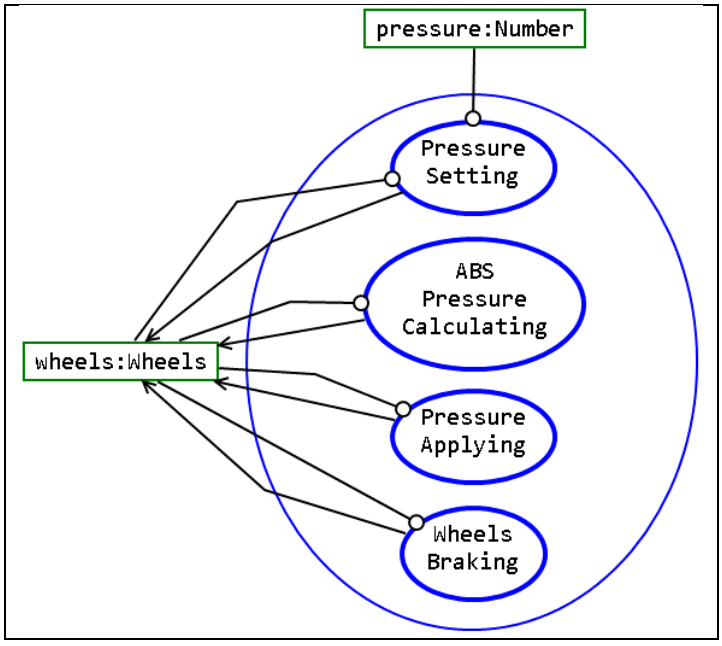


*Figure 122 – **ABS Breaking** in-zoomed process*

Drilling down into the inner processes, **Pressure Setting** takes the pressure that is given by the driver and stores it in each of the wheels, as shown in Figure 123.

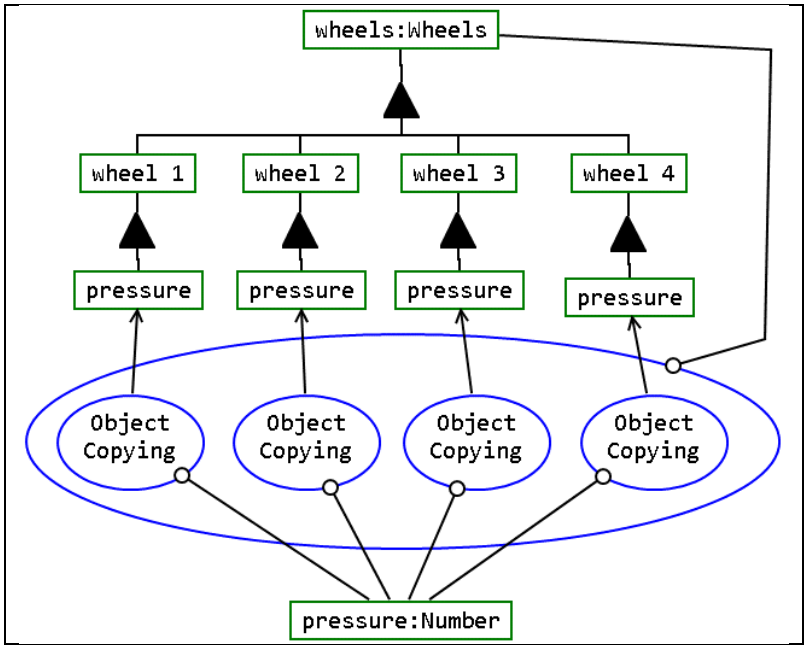


*Figure 123 – **Pressure Setting** in-zoomed process*

Following **Pressure Setting**, **ABS Pressure Calculating**, which is the heart of ABS, is invoked. When a wheel in the car becomes locked (or is in this process), its velocity changes rapidly relative to the velocity of the other wheels. In this case, the breaking pressure for this wheel must be changed so that is does not lock. **ABS Pressure Calculating** does this by calculating the average speed of all wheels, comparing each wheel with this velocity and generating a specific pressure for each wheel, as shown in Figure 124.

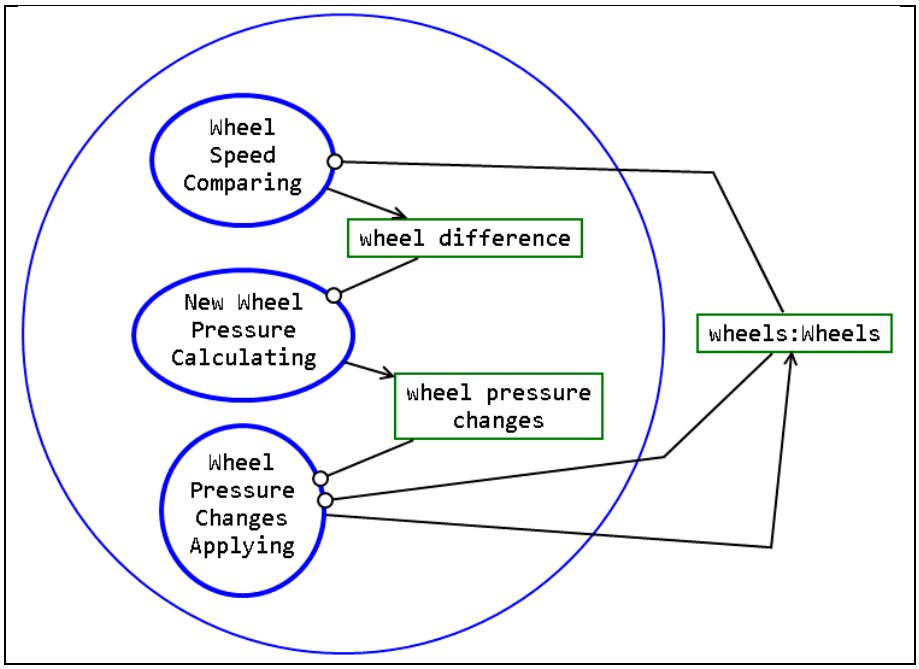


*Figure 124 – **ABS Pressure Calculating** in-zoomed process*

Zooming in Figure 125 into **Wheel Speed Comparing**, we see that it calculates the **average speed** of all **wheels** and then the difference between each wheel and the average. Note here the lack of type for **wheel difference**, demonstrating that typing in OPP is optional.

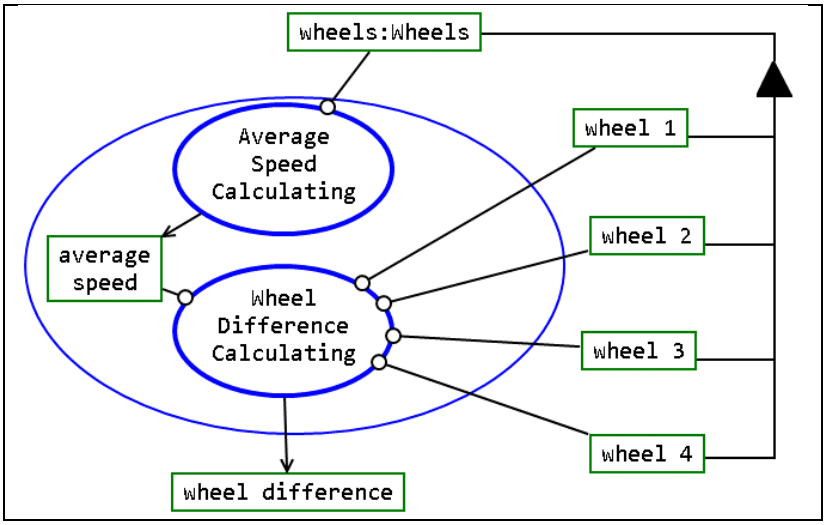


*Figure 125 – **Wheel Speed Comparing** in-zoomed process*

The actual calculation of the average speed is done in **Average Speed Calculating**, shown in Figure 126, summing the speed of the **wheels** and then dividing the sum by 4. While this calculation is trivial, it is shown here to illustrate how OPP can go from high-level programming to low-level programing with basic mathematical operations, such as addition and division.

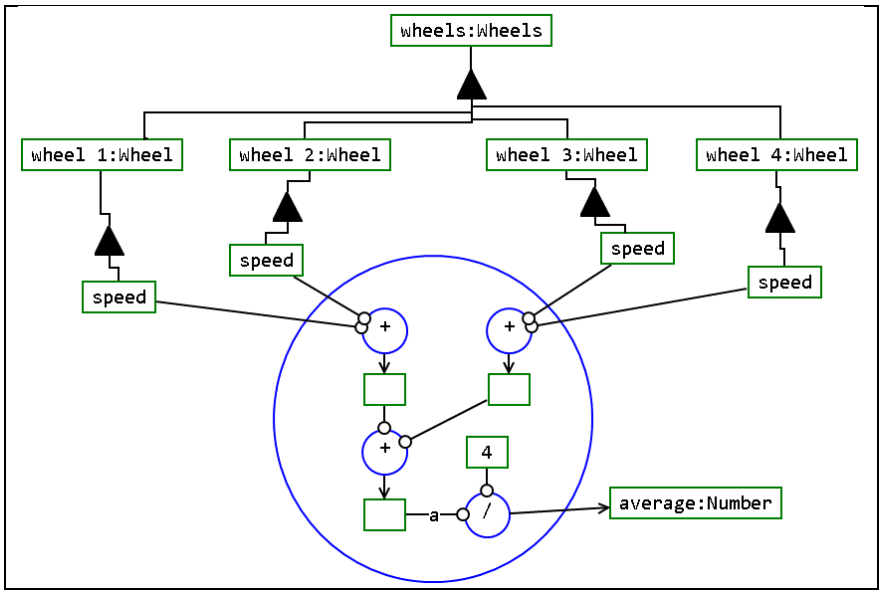


*Figure 126 – **Average Speed Calculating** in-zoomed process*

In a similar way, **Wheel Difference Calculating** in Figure 127 shows the execution of multiple processes that can be executed in parallel – in this case calculating the different between the speed of every wheel and the average speed of all wheels.

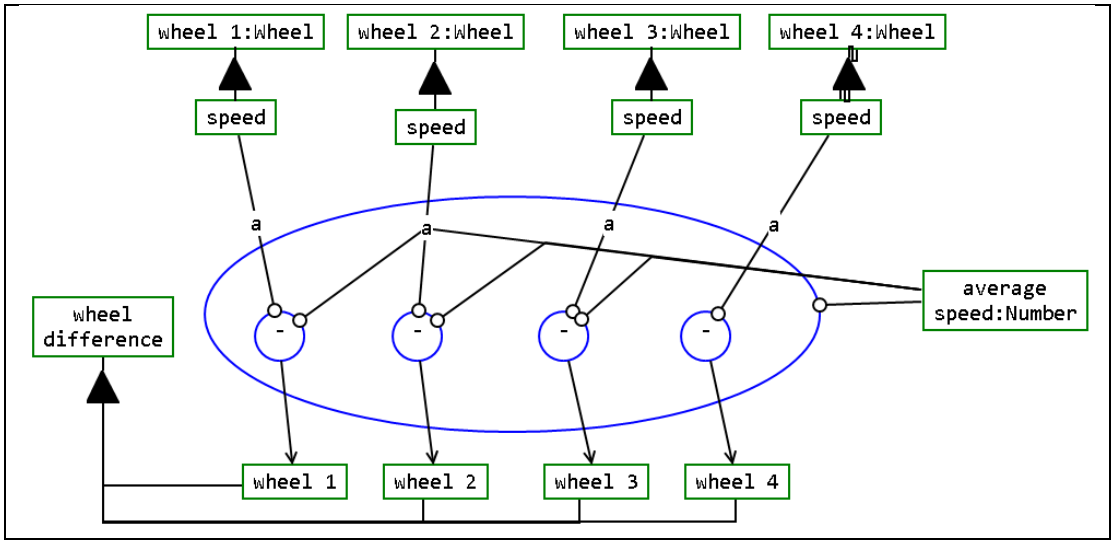


*Figure 127 – **Wheel Difference Calculating** in-zoomed process*

Going back to **ABS Pressure calculating**, it is of interest to zoom into **Wheel Pressure Changes Applying**. In this process, the actual pressure to be applied to each wheel is calculated, as shown in Figure 128.

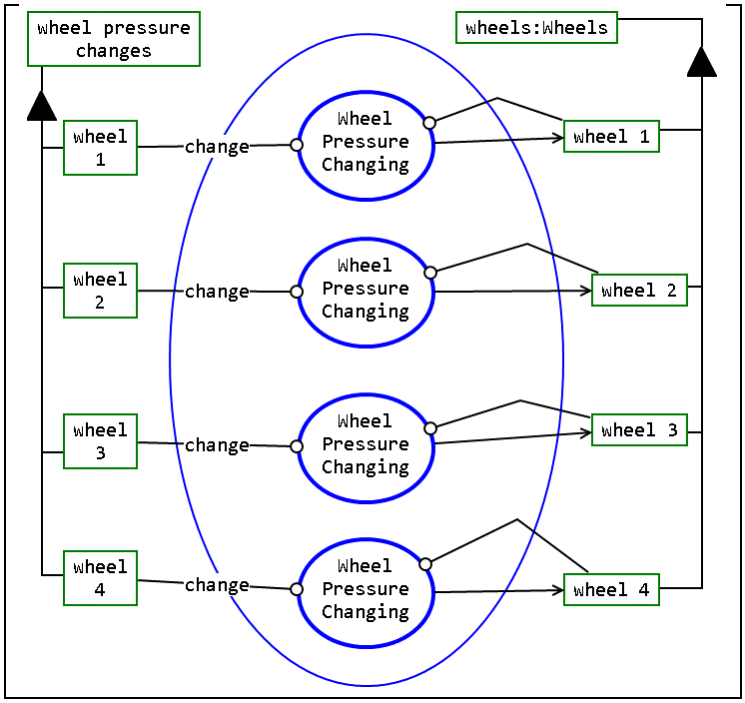


*Figure 128 – **Wheel Pressure Changes Applying** in-zoomed process*

It is also of interest to inspect in Figure 129 the **Wheel Pressure Changing** process, as it demonstrates the use of logical operations and execution flow to define the final pressure that is applied to the wheels by the ABS System. The use of global objects is also shown here with **maximum pressure**, which is a parameter of the system that limits the pressure that can be applied to any wheel.

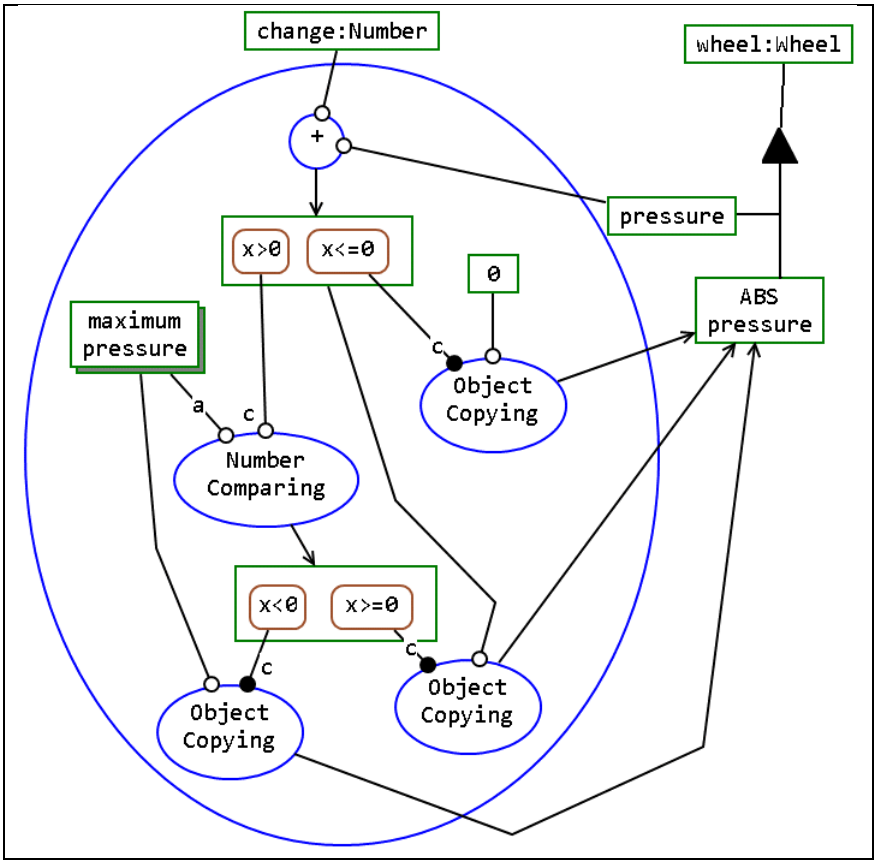


*Figure 129 – **Wheel Pressure Changing** in-zoomed process*

The rest of the in-zoomed processes comprising the system are of similar nature. The full example system is posted in the repository of the editor and interpreter for reference, as described in Section 8.

To test that the ABS program work, we wrote a test process that initialized the ABS system, provided initial values to the wheels, and performed a loop of ABS braking executions until all wheels stopped. Admittedly, this model is a very simplified version of how an ABS system actually works. The **ABS Braking Testing** process is shown in Figure 130.

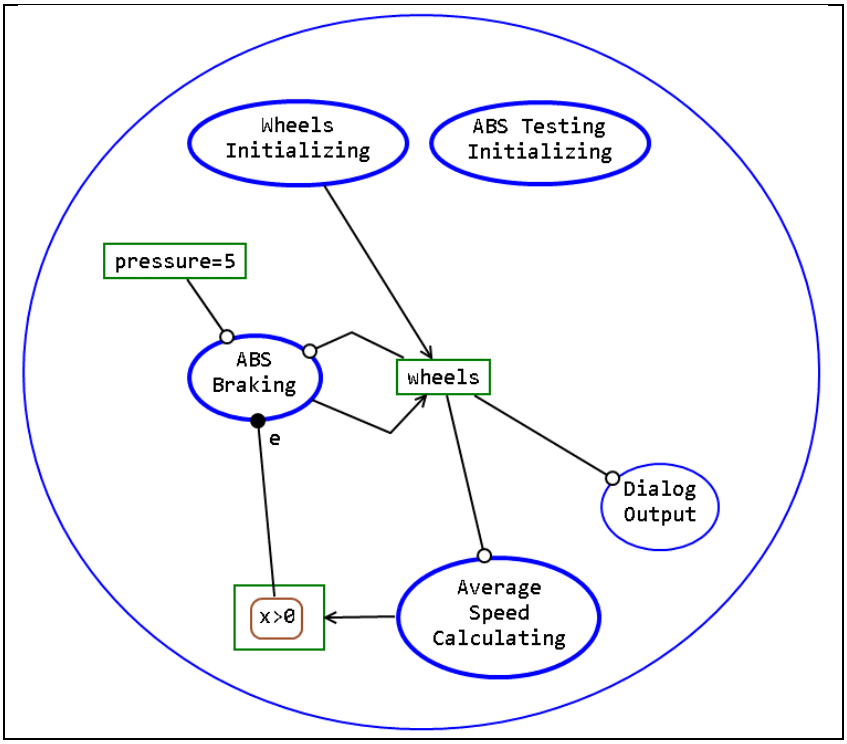
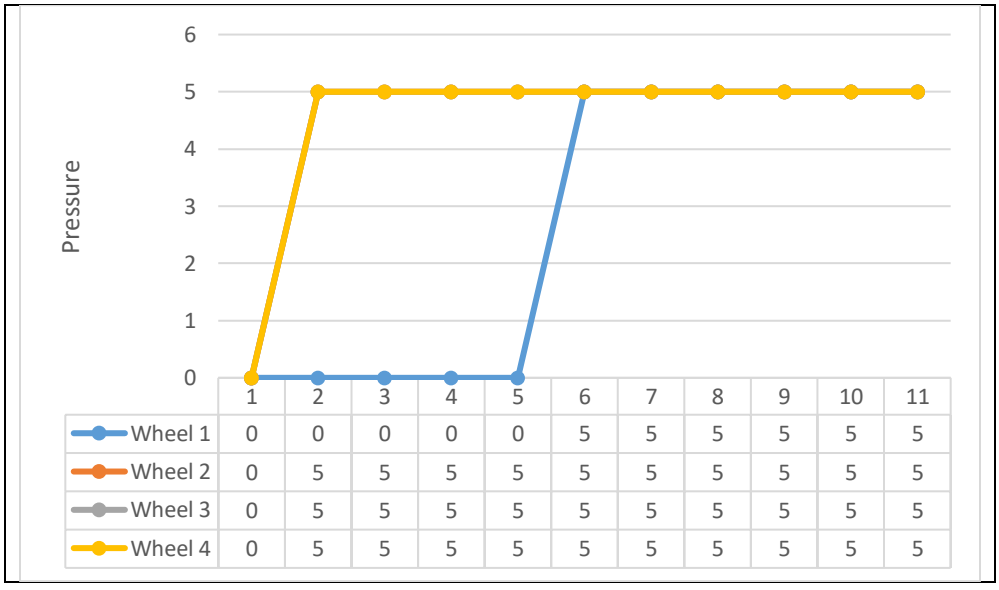


*Figure 130 – **ABS Braking Testing** in-zoomed process*

The wheels were initialized with speed 50, except for one wheel with initial speed of 30, to trigger the ABS braking. The pressure applied to each wheel after the braking happens as a function of time is shown in Figure 131.

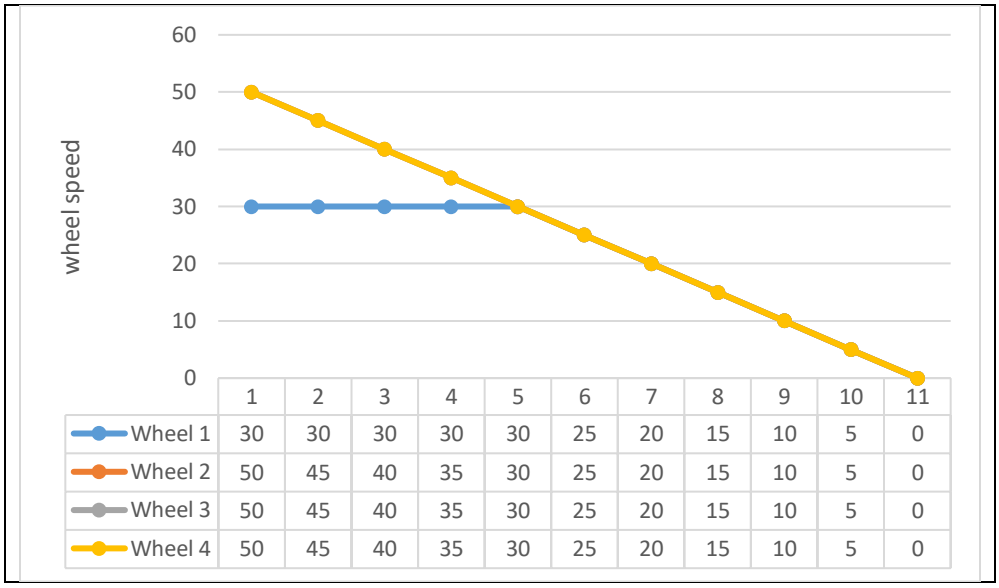


	1	2	3	4	5	6	7	8	9	10	11
Wheel 1	0	0	0	0	0	5	5	5	5	5	5
Wheel 2	0	5	5	5	5	5	5	5	5	5	5
Wheel 3	0	5	5	5	5	5	5	5	5	5	5
Wheel 4	0	5	5	5	5	5	5	5	5	5	5

Figure 131 – Pressure applied to each wheel over time since braking

As can be seen in the graph, during the first 5 time units, the system applies to wheel 1 a different pressure than to the other three wheels, as it has detected that its speed is much

smaller that the speed of the other wheels. This effect can be seen in Figure 132, where the speed of the wheels as a function of time is shown after the braking process is executed.



	1	2	3	4	5	6	7	8	9	10	11
Wheel 1	30	30	30	30	30	25	20	15	10	5	0
Wheel 2	50	45	40	35	30	25	20	15	10	5	0
Wheel 3	50	45	40	35	30	25	20	15	10	5	0
Wheel 4	50	45	40	35	30	25	20	15	10	5	0

Figure 132 - Wheels speed as a function of time

The ability to create an executable model for the system and then test the model by executing it shows the power of OPP as a tool for the development of complex systems, especially in the early stages of the development, when there is no physical prototype and initial simulations can be done "*in silico*".

7 EXPERIMENTATION

To evaluate OPP, two experiments were performed. The first experiment was done with a group of 104 undergraduate students and the second as a focus group discussion with six professional developers and system engineers who are familiar with OPM and various programming languages.

7.1 STUDENT EXPERIMENT

We performed an experiment in which 104 students in an undergraduate information systems engineering course were given a task to program very simple systems using OPP and its development environment. The participants in the course learned the semantics of OPM, and had also taken at least one undergraduate programming course. The participants received a short one-hour tutorial on OPP, and a full online tutorial was made available one week prior to administering the task to the research participants.

The course was divided in groups of 2-3 students who implemented one of the systems. After the programming task, the students were given an electronic self-assessment survey of their understanding of the requirements and the languages and of their opinion on OPP and its development environment. Both the exercise and the survey could be submitted within one week.

Of all the groups that participated in the course, 36% developed a system that conformed to the requirements, 60% developed a system that implemented only part of the requirements, and the remaining 4% returned an empty or completely incorrect program.

The survey, which was carried out around the middle of OPP development, showed that most students (67%) understood the requirements of the task, and more than half of the students (57%) indicated they understood the language sufficiently to perform the exercise. Many of the students (39%) had problems interacting with it, indicating that the development environment had to be improved. Based on this finding, an improved user interface was implemented after the experiment, as elaborated below. Finally, most of the students (67%) thought that OPP cannot be used for real-world programming, as it requires a lot of work to implement simple programs.

Part of the students in the course (12) are also professional developers with an average 2.5 years of professional developer experience. Compared to the general population, these students understood the language better (63% vs. 57% in the general population). However, a larger percentage of these students (76% vs. 67% in the general population) evaluated OPP as having lower value for them as a programming language.

As a result of the experiment, a number of improvements were made to OPP based on the student survey, the systems they built, and informal conversations that the researchers had with the experiment participants:

1. The OPP version used in this experiment did not have different types for **List**, **Set**, and **Complex Object** types, but only one **Complex Object** type which was used for all of them. This caused confusion in the students. Because of this, we decided to create specific types for each as they each fulfill different use cases.
2. The experiment raised a number of usability issues in the OPP editor, including the following: (1) selecting procedural links was difficult because they had to be clicked exactly while their width was very thin; (2) OPDs could not be renamed; (3) Manipulation of structural links was complex and took much effort to make them look good. A new version of the editor created after the experiment fixed many of these usability problems.
3. Using Eclipse as the platform where OPP programs are developed creates user friction as non-technical users are awed by the complexity of the platform, which is targeted mainly to developers. While not possible in the current version, a Web-based version of OPP is planned to be developed as part of the OPCloud project, and we expect that as a result of the improved graphic user interface, which is already under development, the barrier for language acceptance will be lowered.

For an extended description of the experiment, see Appendix 1 in Section 10.

7.2 Experts Focus Group

To get an expert's view of the OPP language, a focus group with six professional developers and system engineers from the Technion's Enterprise Systems Modeling Laboratory was performed. The session began by an explanation of the differences between OPM and OPP, followed by a presentation and a demonstrated execution of the ABS System use case described in Section 6. After this, each member of the focus group gave a summary of her or his view of the language, its strengths and weaknesses, as well as comments related to comparison with other languages. The focus group meeting lasted 1.5 hours, and was audio-taped and summarized. The main insights of the group are the following:

- One of the interesting properties of OPP (and of VPLs in general) is that one can clearly see structures in the program that are hard to see in code. Regularity in code, like clone detection, is something that is difficult to identify. A graphical pattern that repeats itself is detected much faster by the brain, because the visual elements are the same or very similar.
- Since OPP has a very small set of building blocks, it may be easy for new programmers and young programming students to start developing a system, compared to the syntax

of textual programming languages. This point was raised by multiple members of the focus group, and it is an interesting research subject that should be pursued in the future.

- As the language already has a visual representation, creating a "visual map" of the program that enables one to navigate between processes could highly improve the usability and understandability of the language.
- OPP can be useful for system engineers that want to look at a big system and how it behaves. The closer the user is to the final implementer of the system, it seems that OPP becomes less useful. The strengths of OPP are in the possibility of seeing a full system that is also executable, for initial testing and validation, but less so for the low-level implementation of each process, where depicting each arithmetic operation becomes tedious. Indeed, programming low level processes in OPP is not optimal. Multiple participants expressed this opinion. There is an optimal point at which one should switch from visual to textual programming. If the OPP environment can be enhanced with the option to do this switch, it would greatly enhance its usability. When to make this transition is yet another domain for future research.
- There is also a lot of value in using OPP for programmers that are learning a new system, as the understanding of many separate parts of textual code is harder than looking at the system in OPP.
- The ability to define the boundaries of the system in a clear-cut way is very appealing. The high-level diagrams which shows which processes the system executes, which data flows to and from the system is something that is not part of textual programming languages yet is a very important aspect of any system.
- Doing code generation from OPP programs could be useful as a starting point for rapid application prototyping and testing. As round-trip code generation is very hard, this would not be useful in the long term.

The focus group also gave suggestions for developing certain future features that should be added to the language and editor to make them more usable and appealing. These include:

- Automatic layout: since OPP (like OPM) gives semantics to the location of the visual elements in the diagrams, constrained automatic layout algorithm may be applicable as the amount of layout options is limited.
- Runtime visualization: showing how system is executing, what processes are active, and what values are in the objects of the system could greatly improve the understanding of what the system does.

8 CONCLUSIONS AND FUTURE RESEARCH

In this research, Object-Process Programming (OPP) language has been developed, assessed, and improved. OPP is a visual programming language based on the graphical language of OPM – Object-Process Methodology. OPP builds on principles of OPM, which considers stateful objects, processes that transform objects, and relations among them, as the only building blocks required to model and program any system in any domain and at any level of complexity. In OPP, objects define the data types of the system and contain the data that flows in the system, and processes are operations, procedures, functions, or routines that transform object – create objects, consume objects, or change the value of objects. Program complexity is managed via an in-zooming refinement mechanism, which provides for recursively specifying the details of subprocesses. The system starts at a high, abstract level and is refined all the way to the basic arithmetic operations.

Future research is underway to combine OPM with OPP as part of the OPCloud project – a new Web-based environment for OPM modeling, in which OPP will be an integral part, after it is adapted to align completely with a new planned version of OPM ISO 19450, which will be reformulated to include OPP as integral part of OPM by the end of 2018. A new PhD student is already engaged in this research. When accomplished, this environment will be a unique combination of a conceptual modeling language with built-in programming capabilities that are both graphical and textual, enabling the creation of complete information systems from concept to implementation in the same framework.

Another interesting area of research is the understanding of which programming use cases and practices should be done using textual languages and which ones should use visual a programming language, specifically OPP. In other words, the research question is what is the optimal combination of textual and visual programming that maximizes value to programmers and stakeholders who define those systems and programs, yielding the highest quality programs and systems in which these programs are embedded.

We have implemented OPP as an interpreted language on top of Java, with an Eclipse GEF-based editor. You are invited to download a working copy from GitHub at http://github.com/vainolo/Object-Process-Programming.

9 BIBLIOGRAPHY

[1] I. E. Sutherland, “Sketchpad: A man-machine graphical communication system,” in *Proceedings of the SHARE design automation workshop*, 1964, no. 574.

[2] Y. Singh and M. Sood, “Model Driven Architecture: A Perspective,” in *2009 IEEE International Advance Computing Conference*, 2009, no. March, pp. 1644–1652.

[3] OMG, “MDA Guide Version 1.0.1,” no. June. 2003.

[4] OMG, “Object Management Group.” [Online]. Available: www.omg.org.

[5] S. Kent, “Model Driven engineering,” in *Third International Conference on Integrated Formal Methods, IFM 2002*, 2002.

[6] S. W. Liddle, “Model-driven software development,” in *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, 1st ed., D. W. Embley and B. Thalheim, Eds. Springer, 2011, p. 587.

[7] A. Forward and T. C. Lethbridge, “Problems and Opportunities for Model-Centric Versus Code-Centric Software Development,” in *Proceedings of the 2008 international workshop on Models in software engineering - MiSE ’08*, 2008, p. 27.

[8] R. Glass, *Facts and fallacies of software engineering*. 2003.

[9] Y. G. Guéhéneuc, H. Albin-Amiot, R. Douence, and P. Cointe, “Bridging the gap between modeling and programming languages,” *Technology*, pp. 1–56, 2002.

[10] D. Dori, *Object-Process Methodology: A Holistic Systems Paradigm*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[11] ISO, “ISO/PAS 19450:2015 - Automation Systems and Integration - Object-Process Methodology.” 2005.

[12] J. Somekh, M. Choder, and D. Dori, “Conceptual model-based systems biology: mapping knowledge and discovering gaps in the mRNA transcription cycle.,” *PLoS One*, vol. 7, no. 12, p. e51430, Jan. 2012.

[13] P. Soffer, “ERP modeling: a comprehensive approach,” *Inf. Syst.*, vol. 28, no. 6, pp. 673–690, Sep. 2003.

[14] I. Reinhartz-Berger, D. Dori, and S. Katz, “OPM/Web–object-process methodology for developing web applications,” *Ann. Softw. Eng.*, vol. 13, no. 1, pp. 141–161, 2002.

[15] A. Bibliowicz and D. Dori, “Creating Domain-Specific Modeling Languages with OPM/D - A Meta-modeling Approach,” in *Proceedings of the 8th International Joint Conference on Software Technologies*, 2013, pp. 473–479.

[16] D. Dori, I. Reinhartz-Berger, and A. Sturm, “Developing complex systems with object-process methodology using OPCAT,” *Concept. Model. 2003*, pp. 570–572, 2003.

[17] M. M. Burnett, “Visual Programming,” in *Wiley Encyclopedia of Elecrrical and Electronics Engineering*, 1999, pp. 275–283.

[18] B. A. Myers, *Visual programming, programming by example, and program visualization: a taxonomy*, no. April. New York, New York, USA: ACM Press, 1986.

[19] “Visual programming language,” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Visual_programming_language. [Accessed: 09-Feb-2015].

[20] E. Hosick, “Visual Programming Languages - Snapshots,” 2013. [Online]. Available: http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/.

[21] I. Nassi and B. Shneiderman, “Flowchart techniques for structured programming,” *ACM SIGPLAN Not.*, vol. 8, no. 8, pp. 12–26, Aug. 1973.

[22] D. Harel, “Statecharts - A Visual Formalism for Complex Systems,” *Science of Computer Programming*, vol. 8, no. 3. pp. 231–274, Jun-1987.

[23] Carnegie Mellon University, “Alice.” [Online]. Available: http://www.alice.org/.

[24] MIT, “Scratch,” 2014. [Online]. Available: http://scratch.mit.edu/.

[25] National Instruments, “LabView.” [Online]. Available: http://www.ni.com/labview/.

[26] D. a. Scanlan, “Structured flowcharts outperform pseudocode: an experimental comparison,” *IEEE Softw.*, vol. 6, no. 5, pp. 28–36, 1989.

[27] K. Whitley, “Visual Programming Languages and the Empirical Evidence For and Against,” *J. Vis. Lang. Comput.*, vol. 8, no. 1, pp. 109–142, Feb. 1997.

[28] F. P. Brooks, “No silver bullet: Essence and accidents of software engineering,” *IEEE Comput.*, vol. 20, no. 4, pp. 10–19, 1987.

[29] J. V. Nickerson, “Visual programming: Limits of graphic representation,” in *Visual Languages, 1994. Proceedings., IEEE Symposium on*, 1994, pp. 178–179.

[30] D. W. McIntyre, “comp.lang.visual Frequently-Asked Questions (FAQ).” [Online]. Available: http://www.faqs.org/faqs/visual-lang/faq/.

[31] R. Navarro-Prieto, “Are visual programming languages better? The role of imagery in program comprehension,” *Int. J. Hum. Comput. Stud.*, vol. 54, no. 6, pp. 799–829, Jun. 2001.

[32] T. D. Hendrix, J. H. Cross, S. Maghsoodloo, and M. L. McKinney, “Do visualizations improve program comprehensibility? experiments with control structure diagrams for Java,” *ACM SIGCSE Bull.*, vol. 32, no. 1, pp. 382–386, Mar. 2000.

[33] M. Petre, “Mental imagery and software visualization in high-performance software development teams,” *J. Vis. Lang. Comput.*, vol. 21, no. 3, pp. 171–183, Jun. 2010.

[34] S. Stobart, “Use, problems, benefits and future direction of computer-aided software engineering in United Kingdom,” *Inf. Softw. Technol.*, vol. 33, no. 9, pp. 629–636, Nov. 1991.

[35] OMG, “OMG Unified Modeling Language (OMG UML) version 2.3, Infrastructure,” no. May. p. 226, 2010.

[36] OMG, “Model Driven Architecture.” 2001.

[37] D. Thomas, “MDA: Revenge of the modelers or UML utopia?,” *IEEE Softw.*, vol. 21, no. 3, pp. 15–17, May 2004.

[38] R. Gelbard, D. Te’eni, and M. Sade, “Object-Oriented Analysis: Is It Just Theory?,”

Software, IEEE, vol. 27, no. 1, pp. 64–71, 2009.

[39] B. Dobing and J. Parsons, "How UML is used," *Commun. ACM*, vol. 49, no. 5, pp. 109–113, 2006.

[40] R. France and B. Rumpe, "Does model driven engineering tame complexity?," *Softw. Syst. Model.*, vol. 6, no. 1, pp. 1–2, Jan. 2007.

[41] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, "Model-driven development using UML 2.0: promises and pitfalls," *Computer (Long. Beach. Calif).*, vol. 39, no. 2, pp. 59–66, 2006.

[42] D. Dori, "Why Significant Change in UML is Unlikely," *Commun. ACM*, vol. 45, no. 11, pp. 82–85, 2002.

[43] A. Nugroho and M. R. V. Chaudron, "A survey into the rigor of UML use and its perceived impact on quality and productivity," *Proc. Second ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas. - ESEM '08*, p. 90, 2008.

[44] F. Jouault, J. Bézivin, and M. Barbero, "Towards an advanced model-driven engineering toolbox," *Innov. Syst. Softw. Eng.*, vol. 5, no. 1, pp. 5–12, Mar. 2009.

[45] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez, "Modeling in the Large and Modeling in the Small *," *Architecture*, pp. 33–46, 2005.

[46] D. Alonso, C. Vicente-Chicote, J. A. Pastor, and B. Álvarez, "StateML: From Graphical State Machine Models to Thread-Safe Ada Code," *AdaEurope 2008 LNCS 5026*, pp. 158–170, 2008.

[47] J. M. Fernandes, J. Lilius, and D. Truscan, "Integration of DFDs into a UML-based Model-driven Engineering Approach," *Softw. Syst. Model.*, vol. 5, no. 4, pp. 403–428, Jun. 2006.

[48] I. Reinhartz-Berger and D. Dori, "A Reflective Meta-Model of Object-Process Methodology: The System Modeling Building Blocks," in *Business Systems Analysis with Ontologies*, P. F. Green; and M. Rosemann, Eds. 2005, pp. 130–173.

[49] D. Dori, "Words from pictures for dual-channel processing," *Commun. ACM*, vol. 51, no. 5, pp. 47–52, May 2008.

[50] M. Peleg and D. Dori, "Extending the object-process methodology to handle real-time systems," *JOOP*, vol. 11, no. 8, pp. 53–58, 1999.

[51] A. Sturm, D. Dori, and O. Shehory, "Single-model method for specifying multi-agent systems," *Proc. Second Int. Jt. Conf. Auton. agents multiagent Syst. - AAMAS '03*, p. 121, 2003.

[52] D. Dori, R. Feldman, and A. Sturm, "From conceptual models to schemata: An object-process-based data warehouse construction method," *Inf. Syst.*, vol. 33, no. 6, pp. 567–593, Sep. 2008.

[53] A. Bibliowicz and D. Dori, "A graph grammar-based formal validation of object-process diagrams," *Softw. Syst. Model.*, vol. 11, no. 2, pp. 287–302, Apr. 2011.

[54] D. Dori, D. Beimel, and E. Toch, "OPCATeam–collaborative business process modeling with OPM," *Bus. Process Manag.*, pp. 66–81, 2004.

[55] "Inheritance," *Dictionary.com*. [Online]. Available:

http://www.dictionary.com/browse/inheritance?s=t. [Accessed: 29-Apr-2016].

[56] “Gradual Typing,” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Gradual_typing. [Accessed: 22-May-2016].

[57] “Command pattern,” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Command_pattern. [Accessed: 07-Jul-2016].

[58] D. Dori, *Model-Based Systems Engineering with OPM and SysML*. New York, NY: Springer New York, 2016.

[59] “Interpreter,” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Interpreter_(computing). [Accessed: 21-Sep-2016].

[60] ECMA International, “The JSON Data Interchange Format.” p. 14, 2013.

[61] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, “Aspect-oriented programming,” *ECOOP’97—Object-Oriented Program.*, no. June, pp. 220–242, 1997.

[62] E. Foundation, “Eclipse.” .

[63] Wikipedia, “Model-View-Controller.” .

10 APPENDIX 1 – STUDENT EXPERIMENT

The first experiment performed with OPP was done with 104 students in the undergraduate Information Systems Analysis and Design course in the faculty of Industrial Engineering and Management at the Technion – Israel Institute of Technology.

Students in this course have all passed at least one undergraduate programming course. The programming languages known by the students of the course are shown in Figure 133. Only languages known by more than 5 students are shown in this table.

Programming Language	Number of Students
C	58
C++	53
C#	8
Python	18

Figure 133 – Programming languages known by undergraduate students

12 out of the 104 students (11%) considered themselves professional developers, with an average of 2.5 years of experience.

The participants in the course learned the syntax and semantics of OPM during this course and are given projects where they use the language for systems modeling, therefore we assume that their knowledge of OPM is fresh but at the same time at a novice level.

The experiment was performed in the tutorial hours of the class, in 5 different classes. In preparation for the experiment, each tutorial group was given a 30-minute introduction to OPP and its main differences to OPM. The students were also shown how to develop with the OPP development IDE, which differs from the program used by the students to create OPM models. A full online tutorial of OPP was also made available to the students for reference during their work.

The students were divided in groups of 2-3 students. Each group was given one out of 5 different systems, and were given one process to implement in one of the systems. The following 5 systems were given to the students:

- Movie Ratings: a system where users to rate movies.
- Car Wash: a system to manage the business of a car-wash.
- Bakery: a system to manage the business of a bakery.
- EShop: an electronic shopping system.

- Grading: a system to manage student grades.

All system has similar processes to update objects in the system and calculate values based on the objects that existed in the system. The EShop system will be shown here as example.

The description given of the students of the system was: “The system is used by a seller to sell item to buyers through the internet. The system is composed of the following parts (not all parts are shown):”, followed by the System OPD, which is shown in Figure 134.

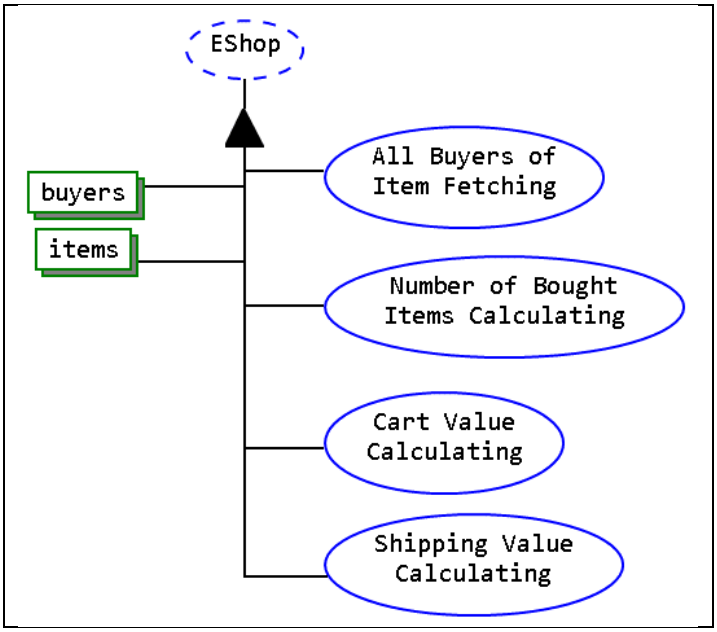
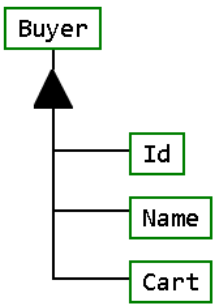
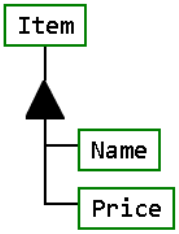
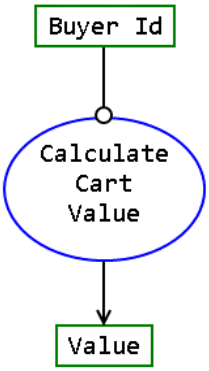


Figure 134 – EShop System OPD

The students were then given the Interface OPD of each one for the processes and the Type OPD for Buyers and Items, parts of which are shown in . To support the visual representation, a textual description of the types and processes was also give.

		
Buyer Type OPD. The global **buyers** object contains **Buyer**	**Item** type OPD. **Item**s are contained in a cart	**Calculate Cart Value** Interface OPD. The process calculates the

objects. The Cart contains **Item**s.		total cost of all items in a **Buyer**'s **Cart**

Figure 135 – Type and Interface OPDs in the EShop system

Grading of the programming task was based on participation in the experiment and not on the result of the program submitted.

After the programming task, students were given an electronic survey to do a self-assessment of their understanding of the requirements and the languages, and to give their opinion on OPP and its development environment. The survey contained the questions shown in Figure 136:

	#	Question	Answer Type
Background Questions	1	How much programming experience do you have	1-3 (1-basic, 3-professional)
	2	How many years of programming experience do you have	Number of years
	3	Which programming languages have you learned	List and option to add more
Understanding of Requirements	4	I understood the description of the system described in the assignment	1-5 (1-disagree, 5-agree)
	5	I understood the different behaviors that the system exposes	1-5 (1-disagree, 5-agree)
	6	I understood the requirements of the process that I had to develop	1-5 (1-disagree, 5-agree)
Understanding of OPP	7	My knowledge of the OPP language was sufficient to perform this assignment	1-5 (1-disagree, 5-agree)
	8	I understand how OPP executes an In-Zoomed OPD	1-5 (1-disagree, 5-agree)
	9	I understand how the OPP interpreter handles data	1-5 (1-disagree, 5-agree)

Usability of the OPP IDE	10	I had no problem interacting with the OPP development environment	1-5 (1-disagree, 5-agree)
	11	I think non-experienced users can use the OPP environment	1-5 (1-disagree, 5-agree)
	12	Having multiple ways to perform an operation, such as adding objects and links, simplifies the interaction with the OPP environment	1-5 (1-disagree, 5-agree)
	13	Having automatic sizing of language elements improved the interaction with the environment	1-5 (1-disagree, 5-agree)
Practicality of OPP	14	While developing the program, I invested a lot of time arranging the elements in the diagram in order for it to look good	1-5 (1-disagree, 5-agree)
	15	OPP requires a large number of OPDs to build a simple program	1-5 (1-disagree, 5-agree)
	16	OPP cannot be used for real programs because you have to work a lot to develop a simple program	1-5 (1-disagree, 5-agree)
Comparison with other programming languages	17	Compared to programming languages that you have used previously, what are the positive aspects of OPP	Free text
	18	Compared to programming languages that you have used previously, what are the negative aspects of OPP	Free text
Comparison with OPM	19	Please list differences that make OPP better than OPM	Free text
	20	Please list differences that make OPP worse than OPM	Free text

Figure 136 – Student experiment questionnaire

Of all the groups that participated in the course, 36% developed a system that conformed to the requirements, 60% developed a system that implemented only part of the requirements, and the remaining 4% returned an empty or completely incorrect program.

The survey showed that most students (67%) understood the requirements of the exercise, and about half of the students (57%) felt they understood the language sufficiently to perform the exercise. It seems that the development environment must be improved since many of the students (39%) had problems interacting with it. Finally, most of the students (67%) think that OPP cannot be used for real world programming, as it requires a lot of work to implement simple programs.

Part of the students in the course (12) are also professional developers with an average 2.5 years of professional developer experience. Compared to the general population, these students understood the language better (57% vs. 63%). On the other side, these students also saw lower value to them language than the general population (76% of the professional developers saw low value to the language, against 67% of the general population).

Analysis of the open questions provided more insights into the positive and negative aspects of OPP. We perform manual analysis of the answers (originally in Hebrew) to extract the main themes written by the students in these answers.

Open Questions – Comparing of OPP with textual programming languages

When asked "Compared to programming languages that you have used previously, what are the positive aspects of OPP", many students (35) said that OPP is easier to understand than textual programming language, especially for non-programmers. A large number (23) also said that the visual representation of OPP makes it more usable. Other topics that rose in the answers where the ease of use of the language and IDE, its simplicity, and readability. Eighteen students found no positive aspects of OPP compared to textual programming languages.

When asked "Compared to programming languages that you have used previously, what are the negative aspects of OPP", most students (31) thought that building real programs with OPP would be too much work, and that is would be too complicated to maintain (23). Other topics that rose in the answers where the time spend making "beautiful" diagrams, the time needed to learn the language, and that it was not intuitive. Twelve students said the IDE of the language was not mature or stable enough.

Open Questions – Comparing OPP with OPM

When asked "Please list differences that make OPP better than OPM", a large number of students (41) stated that the IDE developed for OPP was a great improvement over the current OPM IDE, in many factors such as stability, usability, and performance. Another positive aspect of OPP is its exact semantics (20) and its executability (12). Some students saw the language as simpler than OPM and others also liked the pre-defined functions that came with the language. Eight students saw nothing better in OPP over OPM.

When asked “Please list differences that make OPP worse than OPM”, most students (12) stated that the language was more complex than OPM, that the IDE was less user friendly that the current OPM environment (12), and that the lack of a textual representation (OPL) in OPP was something they missed a lot from OPM (9). Some students said that its software-focused semantics limited it usefulness, and that this required them to do very low level modeling that they didn’t do in OPM. Twelve students saw nothing worse in OPP than OPM.