An Experimental Methodology for Specification Simulators

Vítor Rodrigues[1]

MAP-i PhD student
Faculty of Sciences of University of Porto
vitor.rodrigues@fc.up.pt

Abstract. As the requirements for system analysis and design become more complex, the need for a natural, yet formal way of specifying software systems is everytime more necessary. The Object-Process Methodology (OPM) [PD99] provides a precise modeling language called Object-Process Language (OPL) that supports analysis and design of software systems that is amenable to verification and, potentially, can also serve as a basis for automatic conversion into executable code. However, the views of the project stakeholders can only in part be concretized in analysis and design artifacts and, at the same time, only part of the software system can be derived from the OPM specification models, leaving the final task to the implementers. The absence of a language system capable to unify aspects of requirements specification with the actual software system specification introduces a semantic gap between these two specification languages. The hypothesis of my PhD thesis is that this semantic gap can be minimized by the introduction of a pragmatic component that harmonize aspects of software specification (centered on the semantics of the program) and its implementation (centered on the syntax of the program) and that is capable to combine different programming paradigms such as logic, functional and object orientation.

Keywords: *Object-Process Methodology* (OPM), *Object-Process Language* (OPL), *Test-Driven-Development* (TDD), hybrid design techniques, pragmatics, specification simulators, interpreted programming languages.

1 Introduction

Pragmatics is the study of the use of meaningful strings to communicate about extra-linguistic structure in an interaction process between users of the language. A very concise definition of pragmatics is that syntax studies *Form*, semantics studies *Form + Content*, and pragmatics studies *Form + Content + Use* [vE03].

From the point of view of the management of software projects, this last aspect can be of extreme relevance: if during the initial analysis phase there is a place for referring to the use of already available implementation artifacts (interface signatures), then there is the possibility to analyze the problem already in terms of contracts with these abstract interfaces. One *de facto* development methodology that employs this form of thinking is Test-Driven Development

(TDD) [Amb04] in which the implementation artifacts are created before their concrete implementation.

The word *use* would have two meanings: (1) from the point of view of stakeholders, they can *use* the specification artifacts, where the semantic knowledge is expressed, more freely because the implementation aspects are simple declarative annotations on the semantic model; (2) from the point of view of the implementers, they can *use* the specification artifacts as the high-level view over the program's business logic and focus mainly on the development of context-independent and reusable pieces of software. In this way, the system specification *is* itself the program and is kept consistent with the views of all the participants at all times [Rod05] [RH04].

Nonetheless, the specification artifacts would have to take in consideration the implementation of the program expressed in declarative means. This way of system specification is very different from the one adopted by the "water-fall" methodology: the implementation artifacts are referred in the initial specification instead of being derived from design artifacts trough successive transformations such as in model-driven engineering. Moreover, a completely declarative specification does not impose a single computation paradigm on the system, such as object orientation, since it is associated computation is based on *interpretation* and not *execution.*

The remainder of this paper is organized as follows. Section 2 introduces the main difficulties in the application of the "water-fall" methodologies and gives possible alternatives based on hybrid techniques. Section 3 enumerates the two different points of view of the users of the software product and its implementers, pointing to possible causes for the mentioned semantic gap. Section 4 gives a brief introduction to the semiotic system of the "specification simulator" and presents an example of an abstract program specification. Section 5 describes the dynamic of the development process and the problem of code generation. Finally, Section 6 presents the conclusions.

2 Why do we need an experimental methodology?

One of the fundamental questions we do during testing and program verification is: "Given an implementation unit, can we check its correctness so that it obeys to the initial specification?" [HJG08]. The difficulty in achieving this goal is that both testing and program verification should be made during all the phases of the project and not only at the final stages like in the Verification and Validation stages of the "water-fall" methodology. The adoption of a refined methodology for testing, such as Test-Driven Development, requires a closed-loop development process with feedback branches so that the initial specification is updated in each iteration. By definition, one methodology of this kind explores experimentation in such a way that it is the basis for software construction.

On the other hand, the Object-Process Methodology raises the level of abstraction much higher to the level of the OPL natural language. The syntax of OPL is well defined and unambiguous and provides a firm basis for automated implementation of executable code generation and database schema definition. However, round-trip engineering within the object-oriented code-generation domain is difficult and works well if the source code that implements the derived object classes is *added* only after achieving a stable abstract model [Rod07]. Moreover, "overcoding" can occur since the design models tend to focus on one solution for the problem and not on the problem itself. The main difficulty of this process resides in the fact that the costs associated with detection of design and coding defects grow exponentially along the "water-fall" project stages.

In both cases, software engineering is concerned with techniques useful for the development of effective software programs, where "effective" depends upon specific problem domains. We make the assumption that to answer the question "Is this technique effective" we need a measurement of the relevant attribute. For example, if effective means low cost, then cost of development is a measure. While the OPM methodology leaves the design implementation details from apart, the TDD methodology has a very low semantic value. Nevertheless, one hybrid methodology would be capable to unify the power of abstract design models with the concreteness of implementation artifacts so that the software product is a effective as necessary [RLM08]. The hybrid methodology proceeds by experimentation, integrating context-independent implementation units that acquire a semantic meaning in the context of an application or business process.

The objective of my PhD thesis is to design a new specification language that is at time a programming language. This language is a "glue language" that is used to articulate different artifacts of the user's program, i.e., the data structures are associated with implementation units throughout the interpretation of the "pure" semantics of the specification statements. Therefore, its purpose is to cover all the "holes" in the specification (interaction sequences for which there are no defined patterns).

The input language of the "specification simulator" would support both design and engineering activities. Here, design corresponds to the creative activity of conceiving a novel solution to a unique problem, whereas engineering is the usage of a predefined solution (represented by the signatures of the implementation units) [BB06]. Put in a different perspective, it shall be considered two levels of *design*. At the higher level of abstraction, design is made by the means of a natural language fragment which brings the design activity closer to that of requirements elicitation. Conversely, at the lower level we also design implementation units but, since they are context independent and must be specified using a particular programming language, we consider this design activity to have a stronger engineering facet.

The main objective is avoid the full top-down design approach that exists in the traditional "water-fall" methodologies and to enrich the problem specification with already available pieces of software in a bottom-up manner [PE93]. This choice can only be profitable if the we provide designers with one modeling language that is not constrained to a particular computational paradigm such as object-orientation. A fragment of natural language would be able to model the problem in a way that users themselves can express their views. The outcome of this hybrid design methodology is that the problem/product specification is made of heterogeneous artifacts [RLM08]. From the engineering point of view, the goal is to validate available specifications of implementation units against one possible abstract specification. Indeed, the experimental methodology proposed in [ZW98] focus on the assessment of the effectiveness with which we go through the process of correlating the two different domains of abstract models and concrete implementations.

The pragmatic value of such a language is that the user sees the immediate effect of correlating a design specification with available implementation signatures when he "runs" the program. The compiler for the language can be seen as a "specification simulator" because it interprets the specification instead of starting by generating code from design artifacts, compile them to bytecode and finally execute the program. The "success" of this new language depends on experimental character of the user experience. There will be a single model containing one abstract program that will be complete when we can say, "I tried it, and I like it." [ZW98].

3 The users' experience

Experiments are done when we can manipulate behavior directly, precisely and systematically. The pragmatic value of a specification language that incorporate implementation artifacts is that the semantics of the program can remain unaltered if we change the implementation signatures and, conversely, we can change the semantic context for a specific implementation usage. This type of experimentation has a correlational effect and can be done in a "toy" situation, where events are organized to simulate their appearance in the real world, or in a "field" situation, where events are monitored as they actually happen. Pure semantic aspects and pure implementation aspects of the program coexist in a single model: the variation in the dependent variable(s) is related to the variation of the independent variable(s). For this reason, experiments evaluating models can be designed as "doubleblind" experiments, where the users of the language don't know what the prediction is until after the experiment is done [Pfl94].

This purely declarative description of knowledge allows us to deal with higher levels of abstraction that characterize the problem and the solution space. What works and does not work will evolve over time based upon feedback and learning from applying the ideas and analyzing the results.

In conception methodologies such as OPM, the designers and analysts are responsible for the high-levels models of the system but it is up to the developers to actually implement and test these models. Consequently, what could be a self-evident model for an expert in problem domain can become incomplete or ambiguous for an implementer. On the other hand, a program written in the pragmatic specification language is always as a tentative hypothesis from which the program is drawn out and tested within the scope of its logical or empirical consequences [Bas96].

The scientific method applied to the whole process proceeds with the following steps. A domain theory to explain a given problem is developed using the specification language constructs. Such semantic expressions are written directly in a fragment of natural language and are taken as one hypothesis which is tested against the implementation artifacts. References to these artifacts are also expressed in natural language (see Figure 2). The data collected from this experiments is analyzed to verify or refute the claims of the hypothesis. The "tunning" of the program consists in the adaptation of the implementation artifacts to their semantic context or vice-versa.

4 The language of the "specification simulator"

The heterogeneity of symbols that coexist inside the language of the "specification simulator", some pertaining to semantics of the program (structure and function) and others to the syntax of the program (algorithmic neutral semantics), provide the opportunity to conceive the simulator as a semiotic system [Tav08]. In the Hjelmslev's semiotic model of language, the fundamental differentiation is between *content* and *expression*. The content plane is used to represent the "meaning" of a symbol (sign) and the expression plane refer to the actual effect produced by the manifestation of the symbol (sequence of symbols).

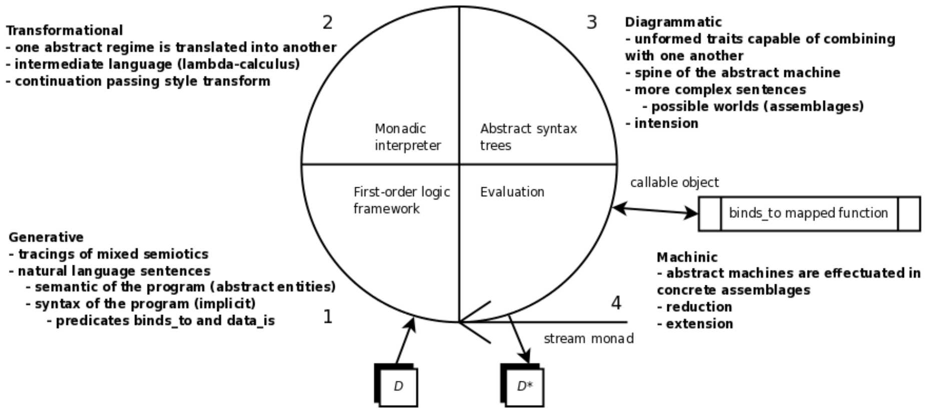
A computer program written using the simulator's pragmatic component is from the beginning dependent on the this "reciprocal" characterization of content and expression, between a domain model and a set of implementation interface signatures. According to Hjelmslev, "there can be no content without an expression, or expressionless content; neither can there be an expression without a content, or content-less expression". The semantic gap between the domain *problem* and one of the *solutions* for it, is precisely the disconnection of the two dimensions [Boy99] and its main cause is that the both specification and implementation languages can conform to different computational paradigms. For example, the OPL language is a textual natural language and it is used to derive object-oriented source code.

The content-expression differentiation can be seen in practice in the work of Hoare [Hoa76] with the *compound* notation. An abstract program can have abstract variables and concrete variables that are processed by a body of procedures or functions. For a given primitive f_j needed by the abstract program, we say that "p_j *models* f_j".

$$t_i \cdot p_j < \textit{actual parameter part} >$$

In order to capture the pragmatic value within the computational effects it is necessary to identify the phases through which the implementation artifacts are transformed. The computational effects are produced by some monadic transformer that is able to bind one data structure to an arbitrary number of functions that operate over the data structure and to produce new values for the same data structure. Wadler proposes monadic functional programming as the appropriate meaning of "Intensional Programming". A monad can be thought of as an operation ($*$) on data types or domains which, given a domain D , yields a new domain $D*$. The idea is that the elements of $D*$ are in some sense generated by the elements of D. The canonical example is to take $D*$ to be the collection of streams over D [Wad].

Figure 1 illustrates four distinct component parts [DG80] of the "specification simulator". This first is the *generative* component that studies the syntactical structure of the natural language sentences and uses a first-order logic framework to express them [DLS93]. The second component is *transformational* and is responsible for transforming the program expressed in first-order logic into the functional programming framework defined by the monadic interpreter [BS06]. The third component is *diagrammatic* because it constructs the diagram of the functional program, that is, the abstract syntax tree of the program that will be evaluated by the monadic interpreter [ADM05]. The fourth component is called *machinic* and corresponds to the evaluation made by the monadic interpreter [Wad95].



2
Transformational
- one abstract regime is translated into another
- intermediate language (lambda-calculus)
- continuation passing style transform
3
Diagrammatic
- unformed traits capable of combining with one another
- spine of the abstract machine
- more complex sentences
- possible worlds (assemblages)
- intension
Monadic interpreter
Abstract syntax trees
First-order logic framework
Evaluation
callable object
binds_to mapped function
Generative
- tracings of mixed semiotics
- natural language sentences
- semantic of the program (abstract entities)
- syntax of the program (implicit)
- predicates binds_to and data_is
1
4
Machinic
- abstract machines are effectuated in concrete assemblages
- reduction
- extension
stream monad
D
D*

Fig. 1. The four components of the pragmatic language system

During the interpretation of a specification statement, a query is made to the first-order logic framework to obtain a reference to the data structures of the abstract entities. These data structures contain the input values for some concrete implementation unit. Using the same query mechanism, it is obtained a callable reference to the implementation unit that "synthesizes" the specification statement. These two implementation artifacts are instantiated in intensional contexts as *possible worlds* of the specification statement and are treated as continuations by the monadic interpreter (see Figure 2). Then, the specification simulator passes the input values to the implementation unit, invokes it and stores the output values back on the data structure. Therefore, there will be different life-cycles for the data and for the implementation units: 1) along time, an XML stream will be created by the simulator containing the history of the computation; 2) the implementation units are instantiated in the scope of a single specification statement.

Example of an abstract program that adds to numbers	
Set of relations	*Entity* A adds to *Entity* B *Entity* A *data_is* A_xml *Entity* B *data_is* B_xml *Relation* adds *mapped_function_is* func_add
Functional program	*Entity* A (*with* A_xml) adds to *Entity* B (*with* B_xml) *in* func_add(type_a, type_b)

Fig. 2. Example of a set of specification statements and the derived functional program

5 Dynamic analysis and simulation

The specification methodology of OPM essentially copes with the initial phase of the development process and the language OPL is "a language that is both formal and intuitive, thereby catering to the need of humans on one hand and machines on the other hand" [PD99]. This appropriateness of the OPL natural language for writing a derivable software prototype is placed in evidence when we consider the *project* life-cycle and the *product* life-cycle. Applying a round-trip process from the specification artifacts to the software prototype, the product life-cycle is also covered by the project's methodology.

The pragmatic language system I propose defines a controlled method that executes the specification, which is the product itself. The consequence of avoiding code generation processes is that the specification is interpreted rather than compiled to some object-oriented programming language. In this sense, pragmatics require a dynamic analysis method [ZW98].

There are two major weaknesses with dynamic analysis. One is the obvious problem that if we instrument the product by adding source statements, we may be perturbing its behavior in unpredictable ways. Also, executing a program shows its behavior for that specific data set, which cannot often be generalized to other data sets. Nevertheless, the underlying experimental method guarantees that the specification of product evolves by small iterations and produces simulation results that are verifiable at all times. The development of the product is cumulative since the very beginning.

From another perspective, the specification model provides the means to evaluate the embedded implementation artifacts. In this case we hypothesize, or predict, how the world views of the domain experts will react to the available technology. The main objective is to model each part of program independently as being either structure, data or functions [JN95]. If we are able to instantiate each of these variables and build an effective simulation environment, results can be obtained more readily than when "water-fall" methods are deployed. By ignoring technological imperatives such as the use of specific computing paradigm or a specific programming language, the simulation of a specification can be easier, faster, and less expensive to run than the "full" product implementation driven by a code-generation process.

6 Conclusions

The motivation for the "specification simulator" is to decouple data structures from implementation units so that data can be stored in XML and manipulated by persistent means and not by memory operations. The link between these two artifacts are the specification statements where the semantic of the program is declared. Data structures are not encapsulated inside an "object" that provides operations like in object-orientation. The relation between a data structure and a function is established only when "interpreting" a specification statement. On the other hand, the implementation units only acquire a semantic meaning during this interpretation.

Despite the fact that OPM provides an effective way to track the product life-cycle since the beginning of its specification, there will be always some doubt if the implementation actually conforms to the specification. I believe that the direct verification of concrete implementations will contribute to reduce this uncertainty without losing the ability to define and benefit from semantic abstractions. Additionally, the introduction of the pragmatic component that articulates aspects of syntax and semantics and the interpretative character of the monadic interpreter can effectively eliminate the semantic gap between models and implementations. In Deleuze's words, "pragmatics is not a complement to logic, syntax, or semantics; on the contrary, it is the fundamental element upon which all the rest depend" [DG80].

In respect to the methodology of writing a program for the "specification simulator", it is based on experiments and collaborative work. A single model aggregates different views of the same problem and the development of the program starts at the same time as its specification. The most significant aspect of this programming paradigm is that the users of the program can, at all times, make amendments to the specification without imposing significant changes on the implementation artifacts or enforcing a delay in the development caused by recurrent source code generation processes. The objective is not to fill the semantic gap introduced by code generation but rather minimize it by including already available implementation procedures in the semantics of the specification language.

References

[ADM05] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theor. Comput. Sci.*, 342(1):149–172, 2005.

[Amb04] Scott W. Ambler. *The Object Primer.* Cambridge University Press, 2004.

[Bas96] Victor R. Basili. The role of experimentation in software engineering: past, current, and future. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 442–449, Washington, DC, USA, 1996. IEEE Computer Society.

[BB06] Ariane Moraes Bueno and Simone Diniz Junqueira Barbosa. Using an interaction-as-conversation diagram as a glue language for hci design patterns on the web. 4385:122–136, 2006.

[Boy99] Nik Boyd. Using natural language in software development. *The Journal of Object-Oriented Programming – JOOP*, 11(9):45–55, 1999.

[BS06] Chris Barker and Chung-Chieh Shan. Types as graphs: Continuations in type logical grammar. *J. of Logic, Lang. and Inf.*, 15(4):331–370, 2006.

[DG80] Gilles Deleuze and Félix Guattari. *A Thousand Plateaus.* Continuum, 1980.

[DLS93] Mary Dalrymple, John Lamping, and Vijay Saraswat. Lfg semantics via constraints. In *Proceedings of the sixth conference on European chapter of the Association for Computational Linguistics*, pages 97–105, Morristown, NJ, USA, 1993. Association for Computational Linguistics.

[HJG08] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Model driven code checking. *Automated Software Engg.*, 15(3-4):283–297, 2008.

[Hoa76] C. A. R. Hoare. Proof of correctness of data representation. pages 183–193, 1976.

[JN95] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. pages 527–636, 1995.

[PD99] Mor Peleg and Dov Dori. From object-process diagrams to natural object-process language. pages 221–228, 1999.

[PE93] Rinus Plasmeijer and Marko Van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[Pfl94] Shari Lawrence Pfleeger. Design and analysis in software engineering: the language of case studies and formal experiments. *SIGSOFT Softw. Eng. Notes*, 19(4):16–20, 1994.

[RH04] Peter Van Roy and Seif Haridi. Concepts, techniques, and models of computer programming. 2004.

[RLM08] Vitor Rodrigues, João Correia Lopes, and Ana Moreira. An hybrid design solution for spacecraft simulators. In *CAiSE Forum*, pages 29–32, 2008.

[Rod05] Vitor Rodrigues. The Role of the Requirements Engineering in Ontology Definitions. Technical report, Master in Informatics course, FEUP, December 2005.

[Rod07] Vítor Rodrigues. On the Specification of Spacecraft Simulators using Object-Oriented Methodologies. Master's thesis, University of Oporto, Departament of Electrical and Computer Engineering, 2007.

[Tav08] Miriam Taverniers. Hjelmslev's semiotic model of language: An exegesis. *Semiotica*, 2008(171):367–394, 2008.

[vE03] Jan van Eijck. Computational semantics with functional programming. Downloaded on March 2004, 2003.

[Wad] Bill Wadge. Monads and intensionality.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.

[ZW98] Marvin V. Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998.