Towards operational natural language.

Alexandr Naumchev

Innopolis University, Innopolis, Russian Federation
Paul Sabatier University, Toulouse, France

Abstract

The multiplicity of software projects’ stakeholders and activities leads to the multiplicity of software specification views and thus creates the need to establish mutual consistency between them. The process of establishing such consistency is error-prone and requires adequate tool support. The present article introduces specogramming – an approach that treats a modern object-oriented integrated development environment as a word processor. The approach turns the process of documenting initial specifications into a simplified form of programming and turns structured-natural-language specifications into runnable programs that yield multiple consistent-by-construction views, one of which is structured natural language.

Keywords: continuous software engineering, specogramming, object-oriented programming, parameterized unit tests, specification drivers, seamless requirements, seamless development

1. Introduction

The multiplicity of specification views leads to the following problems:

1. The problem of producing the views and keeping them in sync.

2. The precedence problem, when two views run out of sync.

3. Reliance on potentially ambiguous structured natural language, when the views’ precedence is not clear.
The following development situation illustrates these problems. It also illustrates the specogramming approach itself in the upcoming sections. Consider a quality assurance (QA) engineer who relies on a unit test view and a developer that relies on a structured-natural-language view, such as user stories. When the QA engineer finds a bug, the QA vs. development conversation begins. The developer does not agree with the unit test used to uncover the bug, which leads to discussing the original user story. Examination of the user story reveals that either the development or the QA engineer has misunderstood the original requirement. The situation results in a waste of time, intellectual, and emotional energy.

The present article introduces specogramming – the process of specifications programming. A specogram is an object-oriented (OO) program that looks like structured natural language. Specogramming treats integrated development environments (IDE’s) as word processors and natural-language texts as programs. Running a specogram results in the generation of the necessary, consistent-by-construction, specification views.

Specogramming solves the three problems above:

1. The problem of keeping the views in sync.
   Changes happen only in specograms, which consistently propagate the changes to all the necessary views.

2. The precedence problem, when two views run out of sync.
   Specograms always have the highest precedence. Running the associated specogram will remove the inconsistency.

3. Reliance on potentially ambiguous structured natural language, when the views’ precedence is not clear.
   In specogramming, a structured-natural-language view is a program. A stakeholder can run this program at any time and see what it means as applied to the views concerning the stakeholder.

Specogramming reconsiders the features of OOP and the supporting IDE’s in the following way:
• In a qualified call “target.call”, the “target” and the “call” represent two word combinations that can follow, in this order, in a natural-language statement.

• When a period symbol is entered, the IDE lists the features available on the target object in accordance with its static type. This is a standard feature of the modern OO IDE’s. Specogramming treats the offered features as possible continuations of the specified phrase.

• Specogramming treats classes as vocabularies. A vocabulary, when instantiated and queried, yields another vocabulary object. The new vocabulary object contains queries with names that are grammatically consistent with the name of the query that yielded the object. The static typing of vocabularies guarantees the grammatical consistency. Properly typed vocabularies guarantee that the compiler will accept only grammatically correct, human-readable specograms.

Specogramming assumes continuous development of new operational vocabularies to keep up with the rapidly growing natural-language vocabularies used for specifying software. A GitHub repository \[1\] contains several vocabularies and examples of their use that should be sufficient for developing the intuition behind the method. The project is currently being developed in Eiffel. Specogramming does not conceptually rely, however, on unique Eiffel’s traits, and applies to any statically typed OO language.

The article illustrates specogramming (Section 3) on a specific example (Section 2), describes the existing specogramming environment (Section 4), and concludes with an outline of the future work and recapitulates the method (Section 5).

2. The gap between structured-natural-language and formal specifications.

The present section formulates questions that motivate the invention of the specogramming approach. Consider the following natural-language requirement, further referred to as “requirement_1”: “a clock tick does not change the clock’s hour if, in the beginning, the minute was smaller than 59”. A little bit more technical variant of this requirement expects basic knowledge of the OO notation from readers: “a clock.tick does not change the clock.hour if, in the beginning, clock.minute < 59.”
The following parameterized unit test (PUT) exercises a candidate implementation of “requirement_1”:

\[
\text{check\_requirement\_1}(c: \text{CLOCK})
\]

\[
\begin{align*}
\text{require} & \quad c.\text{minute} < 59 \\
\text{do} & \quad c.\text{tick} \\
\text{ensure} & \quad c.\text{hour} \sim \text{old} c.\text{hour}
\end{align*}
\]

Calling routine “check\_requirement\_1” with a specific CLOCK instance will test the implementation of the requirement if it meets the routine’s precondition. The inability of the call to meet the precondition denotes irrelevance of the test with respect to the requirement, while the inability to pass the postcondition denotes a bug in the CLOCK implementation. This approach to testing through calling OO representation of ADT axioms is known as parameterized unit testing.

Adding a frame condition, such as “modify(clock)”, to the routine’s specification makes it usable as a driver for specifying the “tick” feature with a contract in the presence of a modular contract-based program prover. The following contract provably meets the “check\_requirement\_1” specification driver, which may be certified with AutoProof, the prover of Eiffel programs:

\[
\begin{align*}
\text{class} & \quad \text{CLOCK} \\
\text{tick} & \quad \text{do} \\
\text{ensure} & \quad \text{old} \text{ minute} < 59 \text{ implies hour} \sim \text{old} \text{ hour}
\end{align*}
\]

The next task is to provide an implementation of “tick” that provably meets the specified contract. Program proving also makes it possible to use verification drivers for checking contracts’ well-definedness. Because of the PUT’s /specification drivers’ verifiability, both dynamic and static, the article uses it as the formal specification notation to illustrate specogramming.
The “check_requirement_1” PUT does not map to the original requirement, although it formally specifies its meaning. Namely, grasping the PUT requires the following, additional, knowledge of:

- The Eiffel syntax.
- The notion of contract.
- The semantics of “does not change” as applied to contracts.

While Eiffel treats contracts as first-class citizens, other languages may not: .NET contracts, for example, look like ordinary instructions inside the routine’s body, which further complicates grasping contracted .NET code.

These complications open the following questions. How to translate a structured-natural-language requirement to a verifiable form, such as PUT’s, so that the translation process:

**Q1** Hides details of a specific programming language (PL)?

**Q2** Hides the underlying contracts?

**Q3** Hides the detailed semantics of intuitively clear natural-language phrases, such as “does not change”, “increment”, “decrement”, and many others?

Specogramming proposes a specific answer to these questions.

### 3. Specogramming

The last modification of “requirement_1” was: “a clock.tick does not change the clock.hour if, in the beginning, clock.minute <59”. Let us continue structuring it: “execution_of “clock.tick” does_not_change “clock.hour” if_in_the_beginning “clock.minute <59”.”. This modification uses the underscore symbol to connect the words related to the requirement’s structure, and quotes the domain-related terminology (the clock terminology in the “requirement_1” example). The next iteration parenthesizes the problem domain-related terminology, and puts the period symbol after each closing parenthesis:
requirement ("requirement_1").states_that.execution_of ("clock.tick").
does_not_change ("clock.hour").for ("clock").of_type ({{CLOCK}}).
    if_in.the.beginning ("clock.minute < 59").period

Figure 1: Application of specogramming to “requirement_1”

This form reflects the main idea behind specogramming: it treats structured natural language as object-oriented executable instructions.

The final form of the requirement adds something else (Figure 1):

- The “requirement ("requirement_1")” call that labels the requirement for traceability.
- The “.for ("clock").of_type ({{CLOCK}})” call adds the typing information about the “clock” variable. The “{{CLOCK}}” expression just yields string “CLOCK” in Eiffel. The advantage of this way of saying “CLOCK” is that the compiler checks if the class exists or not.
- The “.period” command call finalizes the whole instruction by yielding no object on which otherwise it would be possible to do more calls.

Each of the calls, except the “.period” call, yields an object. The static typing of these calls is such that the compiler does not accept structurally invalid requirements. If one forgets to add “.period” in the end, the compiler will remind that it is wrong to have a function call as the last call: one must finalize the instruction with a command call that yields nothing. The compiler cannot, however, rule out malformed inputs to the calls. The calls rule out malformed inputs at runtime, through preconditions: an attempt to run the same instruction as in Figure 1 but with “‘requirement 1’” instead of “‘requirement_1’” will fail: the precondition of the “requirement ()” function requires its input to be a well-formed identifier.
Compiling and running the specogram that contains the instruction in Figure 1 produces the following output:

- A LaTeX document with an entry that turns into the following text when compiled to PDF:

  **requirement 1**: Execution of `clock.tick` does not change `clock.hour` if, in the beginning, `clock.minute < 59`.

  The LaTeX entry resembles the original natural-language requirement with three parts (in italic) formalized.

- A class with the following PUT:

  ```
  check_requirement_1
  -- execution of clock.tick does not change clock.hour
  -- if in the beginning clock.minute <59 :
  -- for any
  (clock:CLOCK)
  -- which
  require
  -- that
  clock.minute <59
  do
  -- executing
  clock.tick
  -- will
  ensure
  -- that
  clock.hour ~ old clock.hour
  end
  ```

  The specogram instruction in Figure 1 produces a PUT that not only contains the required code but also enriches it with human-readable information. The natural-language comments, that start with “--”, make it possible to read the whole “check_requirement_1” routine from the beginning to the end, as a holistic phrase. This is a seamless approach [5, 6, 7, 8, 9] that proposes to interweave the notations, not to switch
between them. Hereafter the article uses term *seamless requirement* [9] to denote such readable-through routines, suitable for both software construction and verification, both dynamic and static.

The following procedure describes the process software development with specogramming as the software development methodology:

1. Write a specogram in IDE.

2. Compile the specogram and fix compilation errors, if any.
   
   a. If a compilation error is caused by the inability of the compiler to recognize `T` in a “`{T}`” expression, declare the type.

3. Run the resulting specogram.

4. Compile seamless requirements resulting from the specogram at step 3.

5. Repeatedly fix compilation errors detected at step 4., if any.
   
   a. If an error talks about non-existing features or classes, create them.
   
   b. Go to step 1. and fix the specogram otherwise.

6. Deploy a verification infrastructure for checking the resulting seamless requirements.
   
   a. Call each of them with arguments that pass their preconditions, if you practice testing [2].
   
   b. Equip implementation classes with contracts that make the seamless requirements pass static verification if you use a static program prover [3].

7. Provide an implementation that passes the checks from the verification infrastructure deployed at step 6.
   
   a. Makes the calls from step 6.a. pass their respective seamless requirements’ postconditions.
   
   b. Is provably correct against the contracts specified at step 6.b.
The implementation phase consists mainly of step 7., but it starts already at 2.a.: a successful compilation of a specogram assumes the existence of all types it talks about. Then, the implementation phase continues at step 5.a.: execution of the specogram turns a string expression of the form “target.call” into the actual call, and if the corresponding feature does not exist, the process requires to at least declare it. Step 5.b. assumes compilation errors caused by the initial specogram; non-declaration of a variable used in a specogram instruction is an example of such an error.

4. Specogramming environment

While the previous sections’ purpose was bringing the intuition behind specogramming, the present section describes the specogramming environment.

The following example represents a complete specogram:

1  specify.software
2  do
3    create specification.further.referred.to.as("clock_specification")
4    requirement("requirement_1").states_that.execution.of("clock.tick").
5      does_not.change("clock.hour").for("clock").of.type([CLOCK]).
6      if_in_the.beginning("clock.minute < 59").period
7    specification.writes.seamless.requirements
8    specification.writes.latex
9  end

1. The instruction on line 3 instantiates a specification object.

2. The instruction spread across lines 4 to 6 adds a requirement object to the specification and specifies the requirement through the chain of qualified calls.

3. Line 7 writes the seamless requirements class. In this example, the class will contain only one seamless requirement, “check_requirement_1” [Section 3].

4. Line 8 writes the latex document. The document will contain only one record in this case:

   requirement_1: Execution of clock.tick does not change clock.hour if in the beginning clock.minute < 59.
The implementation of the “specify_software” routine is readable-through, from the beginning to the end, as a natural-language text.

A specogramming-based project may contain the following clusters (the rightmost pane, “Groups”, in Figure 2):

**Core**: contains implementations of the most important requirements engineering concepts - *requirement*, its hidden *meaning*, and *specification* that consists of requirements. This cluster is supposed to be changed when it is necessary to implement another view or improve existing views’ generation.

**Generated views**: stores specification views produced by specograms.

**Vocabulary**: contains the vocabulary classes used for specogramming. This cluster is supposed to be modified every time a new meaningful vocabulary is found.

5. **Conclusions and future work**

The idea of specogramming has a high potential. The current implementation of the natural-language-like vocabulary performs straightforward generation of structured
specifications with elementary input checks. Nothing prevents enriching the implementation with advanced analysis of the requirements during specograms’ execution. It is possible, in fact, to extend the existing vocabulary for producing and analyzing not only specifications but also implementations. In general, the vocabularies that look like natural language may hide intelligence of unlimited complexity.

Specogramming has the following immediate benefits to software specification practices:

- Taking advantage of the OOP features, such as qualified calls, static typing, and command-query separation, to guarantee requirements’ structural correctness.
- Taking advantage of modern IDE’s’ intelligent features, such as listing the services offered by an object, to facilitate the specification process.
- Taking advantage of the compiler for ruling out both malformed specograms and seamless requirements that they produce.
- Fixing a malformed view happens only in the original specogram, and rerunning it will propagate the fix to each concerned view.
- Wide applicability: all OO languages have qualified calls, the presence of which is the only assumption specogramming relies on.

To make the value of specogramming more evident, we need to do a considerable amount of work:

- Develop methodological recommendations for developing vocabularies; investigate how much this development may be automated.
- Evaluate the approach on a meaningful example (such as Tokeneer [10]).
- Enrich the existing vocabulary with as many meaningful expressions as possible.
- Refine the current design of the solution which may be suboptimal.
- Possibly add traditional contracts to the current set of verifiable specifications produced by specograms.
The Specogram GitHub repository \[1\] includes several examples, including the one used in the present article. Development of this project is happening inside this repository.

References

[1] A. Naumchev, Specogram, a tool for specifications programming, https://github.com/anaumchev/specogram (2017).

[2] N. Tillmann, W. Schulte, Parameterized unit tests, ACM SIGSOFT Software Engineering Notes 30 (5) (2005) 253. doi:10.1145/1095430.1081749
URL http://portal.acm.org/citation.cfm?doid=1095430.1081749

[3] A. Naumchev, B. Meyer, Complete Contracts through Specification Drivers, in: Proceedings - 10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, 2016. doi:10.1109/TASE.2016.13

[4] J. Tschannen, C. A. Furia, M. Nordio, N. Polikarpova, AutoProof: Auto-active functional verification of object-oriented programs, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2015, pp. 566–580.

[5] D. E. Knuth, Literate programming, The Computer Journal 27 (2) (1984) 97–111.

[6] K. Waldén, J. M. Nerson, Seamless object-oriented software architecture, Prentice-Hall, 1995.

[7] B. Meyer, Object-oriented Software Construction (2Nd Ed.), Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[8] B. Meyer, Multirequirements, in: N. Seyff, A. Koziol (Eds.), Modelling and Quality in Requirements Engineering (Martin Glinz Festschrift), MV Wissenschaft, 2013.

[9] A. Naumchev, B. Meyer, Seamless requirements, Computer Languages, Systems & Structures 49 (2017) 119–132. doi:10.1016/j.jcl.2017.04.001
URL http://linkinghub.elsevier.com/retrieve/pii/S1477842416301981
[10] J. Barnes, R. Chapman, R. Johnson, D. Cooper, B. Everett, Engineering the Toke-neer Enclave Protection Software, Proc. of the 1st IEEE International Symposium on Secure Software Engineering (ISSSE) (March).