Design of an Executable Specification Language Using Eye Tracking

Georg Simhandl
University of Vienna, Austria
Faculty of Computer Science
Research Group Software Architecture
georg.simhandl@univie.ac.at

Philipp Paulweber
University of Vienna, Austria
Faculty of Computer Science
Research Group Software Architecture
philipp.paulweber@univie.ac.at

Uwe Zdun
University of Vienna, Austria
Faculty of Computer Science
Research Group Software Architecture
uwe.zdun@univie.ac.at

***Abstract*—Increasingly complex systems require powerful and easy to understand specification languages. In course of the design of an executable specification language based on the Abstract State Machines formalism we performed eye-tracking experiments to understand how newly introduced language features are comprehended by language users. In this preliminary study we carefully recruited nine engineers representing a broad range of potential users. For recording eye-gaze behavior we used Pupil Labs eye-tracking headset. An example specification and simple comprehension tasks were used as stimuli. The preliminary results of the eye-gaze behavior analysis reveal that the new language feature was understood well, but the new abstractions were frequently confused by participants. The foreknowledge of specific programming concepts is crucial how these abstractions are comprehended. More research is needed to infer this knowledge from viewing patterns.**

***Index Terms*—Gaze Behavior, Effects of Language Features, Executable Specification Language, Abstract State Machines**

I. Introduction

Because of the increasing complexity of hardware-software systems, interdisciplinary teams are needed to specify and implement them in a robust and especially efficient way. Specification languages like Systems Modeling Language (SysML) or Unified Modeling Language (UML) enable communication across disciplines, but fall short when it comes to executable models. Executable specification languages try to fill this gap. A novel formal method based Abstract State Machine (ASM) [1] specification language is Corinthian Abstract State Machine (CASM) [2]. Feedback from users, i.e., engineers with various backgrounds in software and/or hardware design and development, is of high importance for the design of such specification languages. As engineers spend more time reading than writing program code, to enable a flat learning curve, and as specification tasks require many stakeholders (including some non-technical stakeholders) to work together, specification languages require a high degree of comprehensibility.

Eye tracking has been used frequently in computer science [3] to investigate human factors of programs and especially established programming or modeling languages. To our knowledge there is no study using eye gaze behavior as a feedback for designing and improving a specification language yet. In this study we focus on eye-gaze behavior in order to investigate recently introduced language concepts of the executable specification language CASM. Our preliminary results of the eye-tracking experiment reveal that the syntax of the language extension is well understood, but surprisingly its structural and behavioral elements are confused. Furthermore the evidence becomes apparent that the foreknowledge of specific programming concepts rather than the programming experience is decisive how new programming language concepts are comprehended.

II. Background

Specification languages support engineers to capture requirements and specify hardware and software systems in an easy and technology independent way. The ASM theory and its formal methods provide the foundation to make specifications executable. The foundational concepts are: (1) an executable *ASM specification* language which looks similar to pseudo code to express rule-based computations over algebraic functions with arbitrary data structures and type domains; (2) a *ground model* serving as a rigorous form of blueprint and reference model; (3) a step-wise *refinement* of the reference model by instantiating more and more concrete models which uphold the properties of the reference model [4]. Despite its potential existing ASM modeling languages do not gain currency. According to Börger [5] there is the need for better abstractions in existing ASM modeling languages to reach the characteristic of a programming language without focusing on class and inheritance concepts. These abstractions should not come at the cost of increasing complexity but remain comprehensible on a high level.

The CASM aims to bridge this gap by providing these language features. Currently we investigate type abstraction with low implementation overhead on *language engineering* side and high understandability on *language user* side. The primary intention behind these new type abstractions is to guide programmers to use exclusive language features and therefore make specifications more comprehensible. Concretely we introduced new syntax definition elements – `structure`, `feature`, and `implement` – to extend the functionality of structures and their behavior similar to *traits* [6], whereas the syntax is influenced by the Rust [7] programming language[1].

[1]https://doc.rust-lang.org/rust-by-example/trait

TABLE I: Participants and Results (JS = JavaScript, PY = Python, VB = Visual Basic)

Participant	Experience			Task 1		Tasks Total		
	Years	Level	Languages	Duration	Correct	Duration	Correct	Difficulty
P1	3	low	Java, JS, PY	22,52s	no	164,57s	75%	5
P2	10	medium	Java, JS, PY	39,58s	no	176,10s	75%	4
P4	19	high	Java, VB, JS, PY, PHP, C++, C#	64,51s	yes	237,60s	63%	4
P5	1	low	Java, C++	32,14s	no	407,42s	13%	5
P6	15	high	C, Java, PY, Haskell, VHDL	22,12s	yes	164,50s	75%	3
P7	5	medium	Java, C++, C#, JS, PHP	26,00s	no	190,34s	38%	4
P8	4	medium	Java, C, C#, C++, Python	39,04s	yes	178,91s	75%	3

In a previous study [8] we compared three different type abstractions *interfaces*, *mixins*, and *traits* in terms of their understandability. The understandability was measured by correctness and response time of the participants performance processing the survey. The results of this previous paper-and-pencil experiment indicate that type abstractions based on *interfaces* and *traits* were more comprehensible than *mixins*.

What is not known so far is how *language users* comprehend these newly introduced structural and behavioral elements and especially how they come to an understanding of these abstractions, i.e. which search patterns are performed while reading and comprehending the specification. In this study we focus on the *traits* syntax and prepared a simple but realistic specification *TrafficLight* (see Listing 1) as a stimulus for the eye-tracking experiment. This sample code extends the basic syntax definition elements from the CASM language[2] [2], namely `function`, `derived` and `rule` by the new `structure`, `feature`, and `implement` definitions.

We hypothesize that eye-gaze behavior can be used to draw conclusions about the the effort, i.e. the cognitive load, which is necessary to understand the specification code in general, and the newly introduced *traits* syntax in particular. Especially we are interested in the *language users'* effort to find and distinguish structural and behavioral elements of the specification. The corresponding research question is: How can eye-gaze behavior help to identify common search patterns and reveal the effort to comprehend the introduced *traits* syntax?

It's commonly known that engineers learn new language abstractions more efficiently the more language paradigms they know. In this study we refer to programming experience as a combination of the time spent with software programming and also the range of different language paradigms. In the context of the main research question, we investigate if and to which extent programming experience influences the effectivity to spot and distinguish structural and behavioral abstractions. The related hypothesis is that *language users* with background in Object-Oriented Programming (OOP) languages with common inheritance concepts distinguish less effectively between structural and behavioral elements, while *language users* familiar with various languages paradigms easily spot the structure and behavior elements.

[2]https://casm-lang.org/syntax

III. Experiment

In order to analyze viewing patterns and visual effort during the process of comprehension of the new language features we set up an experiment to measure eye-gaze behavior. We recorded eye movements with a monocular eye-tracking headset from Pupil Labs[3], equipped with a 200Hz eye camera and a world camera with a resolution of 1280x720 pixels. The *pupil capture* software (version 1.10) was used for recording eye movements and the front facing camera as well as the fixation detection and surface mapping. Subsequently eye-gaze data was mapped to a browser window where stimuli were presented. In this experiment each participant viewed a sequence of assignments described below. As the main stimuli a sample CASM specification was chosen. To let participants directly interact with the specification code, we used the browser-based code editor Monaco[4] supplemented with fiducial markers in the corners of the browser window to facilitate the surface mapping. Instructions and comprehension tasks, similar to works of [9] were presented to the participant in addition to the specification code, while eye gaze and world camera recordings were triggered via the web-socket protocol and the Pupil Lab API controlled by the experiment server. Besides eye and world camera raw recordings, we collected pupil positions, pupil diameter, gaze positions, fixation data and the fixations on surface mappings.

A. Procedure

The procedure of the experiment included (a) the calibration of pupil and eye gaze detection, (b) the presentation of a short introduction to CASM with the basic language features including the newly introduced *traits* concept. (c) When participants hit the *Start* button the recording of eye and world camera was triggered and a brief graphical representation of a system followed. Subsequent task assignments were shown on the top of the window. After pressing the *next* button, (d) the source code of the corresponding specification was shown. As the main stimuli of the experiment we have chosen the *TrafficLight* example specification (see Listing 1) inspired by [10]. Participants were asked to (e) fill-in the missing statements for each of the following tasks: (1) The central component of this system is structure ... (2) The rule ...

[3]http://pupil-labs.com

[4]https://microsoft.github.io/monaco-editor/

```
enumeration Phase = { Stop, Go }
structure Light = {
    function phase : -> Phase initially { Stop }
}
implement Light = {
    derived phase -> Phase = this.phase
    derived oppositePhase -> =
        (if phase = Stop then Go else Stop)
    derived isOn  -> Boolean = (phase = Go)
    derived isOff -> Boolean = (phase = Stop)
    rule switch = {
        this.phase := oppositePhase
    }
}
feature TrafficController = {
    derived lights -> [ Light ]
    derived phases -> [ [ Phase ] ]
    derived position -> Integer
    rule nextPosition -> Integer
    rule control = {
        let currentPhase = phases[ position ] in
        let nextPhase = phases[ nextPosition ] in {
            assert( |currentPhase| = |lights| )
            assert( |currentPhase| = |nextPhase| )
            forall i in [ 1 .. |lights| ] do {
                assert( light[i].phase = currentPhase[i] )
                if light[i].phase != nextPhase[i] then
                    light[i].switch
            }
        }
    }
}
structure OneWayStreet = {
    function lights : Integer -> Light initially {
        1 -> Light(), 2 -> Light() }
    function position : -> Integer initially { 1 }
}
implement TrafficController for OneWayStreet = {
    derived lights -> [ Light ] =
        [ lights( 1 ), lights( 2 ) ]
    derived phases -> [ [ Phase ] ] =
        [ [ Stop, Stop ], [ Go, Stop ]
        , [ Stop, Stop ], [ Stop, Go ] ]
    derived position -> Integer = this.position
    rule nextPosition -> Integer =
        this.position := if position = 1 then 2 else 1
}
```

Listing 1: Executable Specification Code (*CASM*)

defines the main logic of the traffic light signaling. (3) The feature … is implemented … times in this specification. (4) The structure … does not implement a default behavior.

The answers were recorded synchronized with the frame count of the eye gaze recordings. After completion of the experiment (f) a post-hoc interview was conducted to evaluate task difficulty and perceived cognitive load as a ground truth of the measured visual effort [11]. The semi-structured interviews aimed to learn about the participant's professional background and programming experience, concluding with a brief discussion about the experiment and the language itself.

B. Participants

We recruited nine participants (for details, see Table I) with a broad range of software engineering experience, where low in the table corresponds to intermediate level, medium to advanced and high to professional level correspondingly. Four participants were recruited at a German and five at an Austrian university. While all have different professional backgrounds and earned or work towards a computer science degree. All participants volunteered to take part in the experiment. It's

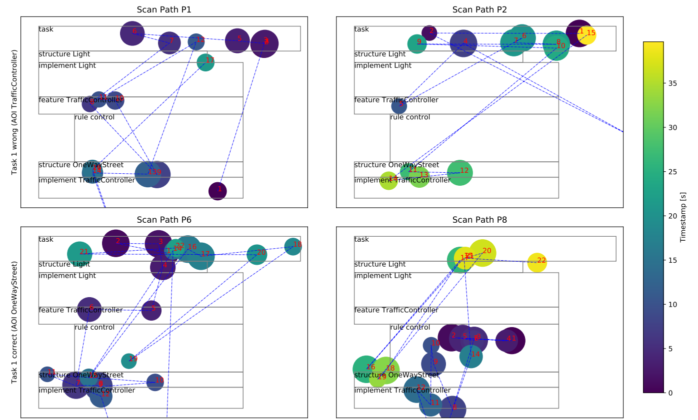

Fig. 1: (a) Fixation positions mapped to specification language elements of participant P1 and P2 whereas the answer is incorrect (b) Fixations for P7 and P8 where answers are correct

noteworthy that all participants - except one (P6) - were new to CASM as this preliminary study especially aimed to investigate the *first impression* of this specification language.

C. Analysis

Eye-gaze positions and fixation positions mapped to the browser window were calculated using Pupil Labs *capture* software [5]. The experiments were conducted in the participant's offices. During two experiments (participants P0 and P3) the front-facing camera of the eye-tracking headset overexposed the white background of the computer screen due to changes in light conditions. These two experiments were excluded from further analysis. Although the calibration was performed for each participant eye-gaze position data was skewed and had to be recalculated by using an individual offset for each participant. Similar to [12] eye gaze positions were translated into character locations in the web based editor to identify code elements of interest for the user. In total six areas representing new language features, i.e. `structure`, `implement`, and `feature` and the task area above the code were selected as main Areas of Interest (AOI). In regard to the research question and to measure the effort to identify and distinguish between structural and behavioral elements of the newly introduced language features, we applied scan path analysis to the eye-gaze data. As shown in Table I four participants (P1, P2, P6, and P8) answered three out of four questions correctly, while P1 and P2 didn't identify the main component of the system, namely *OneWayStreet* but chose the `feature` *TrafficController* instead. In a further step we selected these experiments to contrast the viewing patterns of incorrectly (a) and correctly answered (b) questions. Figure 1 shows the indexed fixation positions represented in bubbles whereas the diameter of the circle represents the fixation length. The color of the fixation marker indicates the start time of each fixation. Furthermore the fixations were mapped

[5]https://github.com/pupil-labs

to the AOIs to facilitate viewing pattern analysis. The analysis reveals independent of the correctness and completion time a common eye-gaze pattern, i.e. eye-gazes moved between the elements `feature` *TrafficController* and `structure` *OneWayStreet* forth and back, right after reading the assignment. Furthermore the relation of programming experience and the comprehension of the *traits* syntax was investigated. As shown in Table I, task completion time is not correlated to the programming experience level. While the eye-gaze behavior indicate the experience level, i.e. typical line-by-line reading for novice and source code skimming for professional engineers, viewing patterns explain - to a certain degree - the causal relationship between task correctness and proficiency in language paradigms beyond script languages and OOP. The interviews were analyzed in two ways. The structured data is shown in the Table I. The summary of post-hoc open discussion was processed to cluster inferential information and compiled in the results section.

IV. RESULTS

Eye-gaze behavior analysis, in general, can not only be used effectively to conclude about the effort in understanding new language designs, but reveals specific issues in comprehension of newly introduced concepts. In particular the preliminary results of the scan path and fixation analysis indicate that there is a confusion of `feature` and `structure` supported by the common pattern where participants eye-gaze fixations alternate between the behavioral and the structural elements, exemplified by participant P1's insecurity to choose either *TrafficController* or *OneWayStreet*. The surprising result of this preliminary study is that `feature` is understood as a structural element while it's designed as a behavioral element. *Language users* with experience in OOP languages are commonly used to read the *interface* entirely as structural and behavioral elements can occur at any point in this type of abstraction. In contrast the *traits* syntax clearly distinguishes between structural and behavioral elements. Hence, participants familiar with multiple language paradigms spot these abstractions more effectively, see Figure 1, scan path for participant 6 for example. The results of the post-hoc interview complement the findings and act as a ground truth: The `feature` abstraction was perceived as a *new thing*. Furthermore the usage of `assert` and also `enumeration` in this context were somewhat surprising for several participants. The majority of participants found that the preparation time was too short, and suggested to provide printed language instructions or online help to learn about language features. Although we presented a very simple code editor some participants reported distractions by the editor itself, e.g. the scroll bar. Finally the post-hoc interviews revealed that participants particular familiar with the *state machines* concept perceived the tasks as highly comprehensible and easy to complete.

V. CONCLUSION AND FUTURE WORK

While specification language comprehensibility can be evaluated by expert interviews, eye-tracking reveals valuable and deep insights into the new language features and related distractions resulting in high cognitive load. The viewing pattern analysis indicates that the *traits* syntax is understood well by programmers with at least intermediate programming skills, but the newly introduced language features `feature` and `structure` are frequently confused. As a consequence it's worth to repeat the experiments using a more sounding keyword for the *traits* syntax, e.g. `behavior` instead of `feature`. As the preliminary results of the study indicate the knowledge of specific software engineering concepts such as *state machines* of *traits* syntax seem to be crucial for the comprehensibility of a new specification language feature. The understanding of viewing patterns related to this foreknowledge can not only help to design programming languages targeted to a specific user group, but help to improve the learning curve by e.g. custom tooltips in Integrated Development Environment (IDE) applications.

The future work, therefore, will not only include the repetition of the experiment with more participants, the extension of the experiment by introducing code authoring tasks and the improvement of fixation and surface tracking algorithms, the gaze-to-code mapping algorithm as well as the experimental setup itself, but also the investigation of (real-time) viewing pattern analysis as feedback for language designers and prospectively also as an extension of an IDE to support engineers to learn and/or apply executable specification languages effectively.

REFERENCES

[1] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide - Specification and Validation Methods," pp. 9–36, New York, NY, USA: Oxford University Press, Inc., 1995.

[2] P. Paulweber, E. Pescosta, and U. Zdun, "CASM-IR: Uniform ASM-Based Intermediate Representation for Model Specification, Execution, and Transformation," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018*, Lecture Notes in Computer Science 10817, pp. 39–54, Springer, 2018.

[3] U. Obaidellah, M. Al Haek, and P. C.-H. Cheng *ACM*.

[4] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Science, 2003.

[5] E. Börger, "Why programming must be supported by modeling and how," in *International Symposium on Leveraging Applications of Formal Methods*, pp. 89–110, Springer, 2018.

[6] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable units of behaviour," in *European Conference on Object-Oriented Programming*, pp. 248–274, Springer, 2003.

[7] N. D. Matsakis and F. S. Klock II, "The rust language," in *ACM SIGAda Ada Letters*, vol. 34, pp. 103–104, ACM, 2014.

[8] P. Paulweber and U. Zdun, "On the Understandability of Type Abstractions in Abstract State Machines: A Controlled Experiment," in *(under review)*.

[9] M. Bielikova, M. Konopka, J. Simko, R. Moro, J. Tvarozek, P. Hlavac, and E. Kuric, "Eye-tracking en masse: Group user studies, lab infrastructure, and practices," *Journal of Eye Movement Research*, vol. 11, no. 3, 2018.

[10] E. Börger and A. Raschke, *Modeling Companion for Software Practitioners*. Springer, 2018.

[11] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *Proceedings of the 26th Conference on Program Comprehension - ICPC '18*, (New York, New York, USA), pp. 286–296, ACM Press, 2018.

[12] A. Begel and H. Vrzakova, "Eye movements in code review," in *Proceedings of the Workshop on Eye Movements in Programming*, p. 5, ACM, 2018.