Specifying with Interface and Trait Abstractions in Abstract State Machines: A Controlled Experiment

PHILIPP PAULWEBER*, GEORG SIMHANDL, and UWE ZDUN, University of Vienna, Faculty of Computer Science, Research Group Software Architecture, Austria

Abstract State Machine (ASM) theory is a well-known state-based formal method. As in other state-based formal methods, the proposed specification languages for ASMs still lack easy-to-comprehend abstractions to express structural and behavioral aspects of specifications. Our goal is to investigate object-oriented abstractions such as interfaces and traits for ASM-based specification languages. We report on a controlled experiment with 98 participants to study the specification efficiency and effectiveness in which participants needed to comprehend an informal specification as problem (stimulus) in form of a textual description and express a corresponding solution in form of a textual ASM specification using either interface or trait syntax extensions. The study was carried out with a completely randomized design and one alternative (interface or trait) per experimental group. The results indicate that specification effectiveness of the traits experiment group shows a better performance compared to the interfaces experiment group, but specification efficiency shows no statistically significant differences. To the best of our knowledge, this is the first empirical study studying the specification effectiveness and efficiency of object-oriented abstractions in the context of formal methods.

CCS Concepts: • **Software and its engineering** → **Formal methods**; **Specification languages**; • **General and reference** → **Empirical studies**.

Additional Key Words and Phrases: Empirical Software Engineering, Controlled Experiment, Specification, Effectiveness, Efficiency, Language Constructs, Interfaces, Traits, Abstract State Machines, CASM

ACM Reference Format:
Philipp Paulweber, Georg Simhandl, and Uwe Zdun. 2021. Specifying with Interface and Trait Abstractions in Abstract State Machines: A Controlled Experiment. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 47 (July 2021), 31 pages. https://doi.org/10.1145/3461694

1 INTRODUCTION

In 1993, Gurevich [24] described the Abstract State Machine (ASM) theory, which is a well-known state-based formal method consisting of transition rules and algebraic functions. It has been used extensively by scientists for a broad research field ranging from software, hardware and system engineering perspectives to specify,

*Corresponding Author

Authors' address: Philipp Paulweber, philipp.paulweber@univie.ac.at; Georg Simhandl, georg.simhandl@univie.ac.at; Uwe Zdun, uwe.zdun@univie.ac.at, University of Vienna, Faculty of Computer Science, Research Group Software Architecture, Währingerstraße 29, 1090, Vienna, Austria.

analyze, verify, validate, and construct systems in a formal way [60]. ASMs are used to formally describe the evolution of function states in a step-by-step manner[1] and are used to specify sequential, parallel, concurrent, reflective, and even quantum algorithms. Based on the ASM theory by Gurevich [24], several theory improvements and ASM-based language implementations were developed, which were summarized by Börger and Stärk [8] and Börger and Raschke [7]. The diversity of ASM-based applications ranges from formal specification of semantics of programming languages, such as those for Java by Stärk et al. [72] or Very High Speed Integrated Circuit Hardware Description Language (VHDL) by Sasaki [63], compiler back-end verification by Lezuo [41], software run-time verification by Barnett and Schulte [3], software and hardware architecture modeling e.g. of Universal Plug and Play (UPnP) by Glässer and Veanes [22], to even Reduced Instruction Set Computing (RISC) designs by Huggins and Campenhout [31].

Nowadays, there are several ASM language syntax definitions and tool implementations available like AsmetaL [20], AsmL [26], Corinthian Abstract State Machine (CASM) [42], and CoreASM [19]. AsmetaL and CoreASM offer a rich tool set to analyze and model ASM specifications and provide a Java-based interpreter to execute and simulate the ASM models. AsmL and CASM are compiler oriented language implementations and offer code generation support of modeled ASM specifications. AsmL is based on the .NET framework whereas CASM provides C/C++ code generation and a high performance interpreter as well. Besides the mentioned ASM languages and tools there exists AsmGofer [65] and eXtensible ASM (XASM) [2], but those projects are discontinued.

In addition, many other state-based formal methods besides ASMs exist with their own languages and associated tools e.g. Alloy [32], DEVS [12], EFSM [10], Event-B [1], STATEMATE [28], Temporal Logic of Actions (TLA) [39], Vienna Development Method (VDM) [5], and Z [57].

1.1 Problem Statement

For various ASM languages and tools, as well as in most other state-based formal methods, the proposed modeling languages lack easy-to-comprehend abstractions for describing structural and behavioral aspects of specifications in a reusable and maintainable manner. Most of today's specification languages have implemented basic object-oriented abstractions such as classes and inheritance. As there are known problems in such abstractions, leading to complexity, ambiguity, and low comprehensibility, such as the *diamond inheritance problem of multiple inheritance* [46], it would make sense to study more advanced abstractions as well. Today, many modern language implementations restrict class-based language constructs to allow only single inheritance models and add additional abstractions such as interfaces [9] or traits [64] to the language. A prominent example for ASMs is the modeling language AsmL [26] which uses the class abstraction along with a single inheritance model to encapsulate the state and behavior. A similar approach can be observed in the state-based formal methods community. Object-Z [69] or Z++ [40] provide class-based language constructs with inheritance and polymorphism concepts.

But it is unclear if insights from modern object-oriented programming languages can be transferred to state-based formal specification languages, as those two kinds of languages are substantially different. For example, a specification language should be rigorous, simple, and self-explanatory, which is not the case for

[1] The ASM theory was formerly called *Evolving Algebra*.

many modern programming languages. Therefore, we aim at empirically investigating how a language user performs by only using one object-oriented abstraction, namely interfaces or traits.

There is a debate in the object-oriented community[2], which of the abstractions, interfaces or traits, is best suited to express behavioral aspects, and many implementations combine different language constructs. A notable example would be the programming language Scala [49], which offers a trait syntax that is similar to the Java [58] interface syntax and offers a class-based implementation and extension syntax. Another example of mixed language constructs, namely interfaces and traits, can be found in the programming language Rust [47], where the language user has to express interface definitions through traits. Empirical research on language constructs in ASM languages and similar state-based formal methods can provide some decision guidance to language designers and compiler engineers on choosing language constructs in specification language designs and implementations. So far such empirical research is rare. Höfer and Tichy [29] analyzed 133 reviewed articles of the Journal of Empirical Software Engineering in the timescale from 1996 to 2006. They have discovered that controlled experiments about formal methods in general are underrepresented and that *"studies about programming languages and programming paradigms are conspicuously absent"*. They further concluded more experiments in this direction would encourage more discussions on the comprehensability of programming languages and formal methods, and eventually improve the language engineering process.

Due to the fact that so far studies about state-based formal methods and the comprehensibility of object-oriented abstractions and language constructs in their context are missing (see Section 2.5), our study also aims to make a contribution to improve the state of empirical knowledge about formal specification languages. Prior to this work, we already have conducted another study [55] and investigated the effects on how language users (experiment participants) understand structural and behavioral aspects of a state-based formal method language (ASM) by reading a given ASM specification as stimuli and answering questions about the properties of given specifications. The provided ASM specifications were represented in three different language constructs – interfaces, mixins, and traits.

1.2 Research Objectives, Hypotheses, and Results

In this empirical study **we investigate which of the object-oriented abstraction syntax extensions – interfaces or traits – is easier to use by a participant while comprehending an informal textual description and modeling a corresponding specification with a certain textual language representation** in the context of state-based formal methods.

State-based formal methods and their modeling languages are usually based on core concepts that are significantly different from classes and objects. Reusable and maintainable specifications would be highly useful in these methods and languages, too, and are largely missing in today's methods and languages. In our study, we use ASMs as a representative of state-based formal methods, and the modeling language CASM [42] [43] [56] [52] as a representative for ASM-based languages and tools. As our study is focused on the general notion of adding object-oriented language constructs to these languages and tools, we believe most of our results can have an impact on other ASM languages. In this study the term **specification effectiveness** corresponds to how well (reading, understanding, and writing) and the term **specification**

[2]See, e.g. https://stackoverflow.com/questions/9205083.

efficiency corresponds to how fast (duration time of processing) a participant comprehends a given stimuli and specifies an example ASM specification using one of the two object-oriented abstractions. We define the experiment goal using the Goal Question Metric (GQM) template [74] as follows: **Analyze** the *Interfaces* and *Traits* object-oriented abstractions (language constructs) **for the purpose of** their evaluation **with respect to** their *specification effectiveness* and *efficiency* **from the viewpoint of** the novice software developer or designer **in the context (environment) of** a moderately advanced university software engineering course. Our hypotheses are influenced by the debate in the object-oriented communities which seems to favor traits over interfaces. We hypothesized that specification effectiveness measured by the dependent variable correctness shows a significantly better performance for traits compared to interfaces as well as that specification efficiency measured by the dependent variable duration shows a significantly better performance for traits compared to interfaces. This hypothesis was influenced by the debate in the object-oriented community, which often discusses traits more favorably than interfaces[3] or points out that "Traits are Interfaces"[4] with code-level reuse functionality. However, it is not obvious whether or not such opinions yield a statistically significant difference, and whether or not they can be mapped to the domain of state-based formal languages. In addition, interfaces are probably the best known abstraction to developers today, and like most ordinary developers our participants are trained in programming languages offering the language construct interfaces in Java or how to model interfaces through a C++ abstract class.

For those reasons, it was interesting to perform the empirical study presented in this paper. The obtained results in this study indeed indicate that the language construct traits show far better understanding compared to interfaces.

1.3 Structure of this Article

In Section 2, we describe object-oriented abstractions, ASMs, the used ASM-based language representations used in this study, and present related studies. Section 3 elaborates the planning of this study. In Section 4, we describe the execution of the experiment, while the results are presented in Section 5 and discussed in Section 6. We conclude the article in Section 7.

2 BACKGROUND

This section discusses some properties regarding object-oriented abstractions, ASMs, and ASM-based language constructs that are of interest in this study. Readers already familiar with object-oriented abstractions, ASMs, and the discussed language abstractions and their corresponding representations may consider to skip some parts of this section.

2.1 Object-Oriented Abstractions

Interfaces define a *protocol* of (typed) operations (signatures) to which an *implementer* of a certain interface (type) must conform [9]. An interface defines a type signature. No behavioral or state information can be defined through interfaces. Each implementer of the interface has to provide an implementation of the complete interface. *Traits* are similar to interfaces with the difference that they can define *stateless behavior* which depends only on the trait itself [64]. Therefore, each implementer can reuse and rely on existing

[3] See, e.g. https://stackoverflow.com/questions/9205083.
[4] See, e.g. https://blog.rust-lang.org/2015/05/11/traits.html.

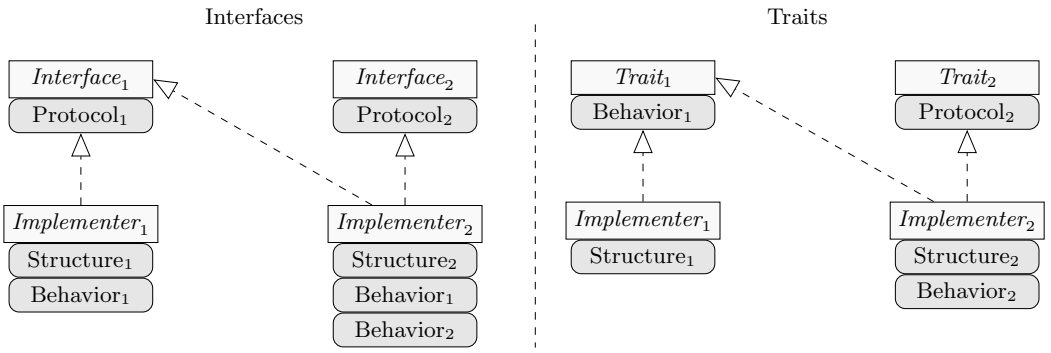


Fig. 1. Overview of Language Construct Properties

behavioral implementations which is not possible through *Interfaces*. Figure 1 depicts both object-oriented abstractions and exemplifies the language construct properties. On the left side, an *Interface* example with two interfaces is shown. $Interface_1$ gets implemented by $Implementer_1$ and $Implementer_2$, whereas $Interface_2$ is only implemented by $Implementer_2$. The same scenario is expressed through the object-oriented abstraction *Traits* on the right side of the figure. As traits can define not only a protocol, the $Trait_1$ directly defines $Behavior_1$ in the trait itself. Thus $Behavior_1$ can be reused by both implementers.

2.2 Abstract State Machines

ASMs are used to express calculations in an abstract manner for many different application fields. According to Gurevich and Tillmann [27], the ASM thesis states that if there is a computer system A, it can be simulated in a step-by-step manner by a behaviorally equivalent ASM B. The resulting ASM theory and formal method consist of three core concepts: (1) an *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; and (3) incremental *refinement* of the reference model by instantiating more and more concrete models which uphold the properties of the reference model [8].

ASMs has two fields of works – *modeling* and *refinement*. In order to model an application or system through an ASM specification, an ASM *language user* has to understand the three most important modeling concepts [7] of ASMs:

States are the notion in ASMs to define the objects and attributes of an application or system through relations and function types. Therefore, all state information in an ASM specification is expressed through a *function* definition (see Section 2.3).

Transactions describe under which conditions the modeled *states* evolve (value change). The evolving is expressed through *transaction rules*. ASMs define several kinds of rules (conditional, iterative etc.) but the most important one is the *update* rule. An *update* rule in ASMs defines which state (function location) shall be updated with a new value. More than one *update* during a transaction is collected in a so called *update-set*. Since ASM rules allow interleaved parallel and sequential execution semantics [25], a *correct* ASM specification does not allow the *update* (insertion to the *update-set*) of

```
function counter : -> Integer // variable

function personsAge : String -> Integer // hash-map
```

Listing 1. *Function* Definition Example

```
derived nextCounter -> Integer = counter + 1

derived isFullAged( name : String ) -> Boolean =
  (personsAge( name ) >= 18)
```

Listing 2. *Derived* Definition Example

```
// named rule
rule incrementOrResetCounter =
{
  // conditional rule (if-then part)
  if nextCounter != 10 then
    // update rule
    counter := nextCounter
  // conditional rule (else part)
  else
    // update rule
    counter := 0
}
```

Listing 3. Named *Rule* Definition Example

the same *function location* twice or more with a different value, which is referred in the literature as an *inconsistent update* [7]. A *language user* can model transactions though *named rule* definitions (see Section 2.3).

Agents are the actors of an ASM specification. There can be one (single) *agent* or multiple *agents*. Every *agent* triggers its top-level *rule* and applies the collected *updates* after the *rule* termination to the *states*. This is called an ASM *step*. Multiple ASM *steps* of one or multiple *agents* form the notion of an ASM *run*, which ends depending on the termination condition modeled in the ASM specification.

Refinement of a modeled ASM specification can be achieved by one of the three kinds – *data*, *horizontal*, or *vertical* refinement. A *data refinement* replaces abstract operations with refined operations which have a one-to-one mapping (e.g., change or make a type more concrete). A *horizontal refinement* makes upgrades to functionalities or changes the environmental settings. A *vertical refinement* adds more details about the application or system (e.g., adding another requirement, more states etc.).

A more detailed description and elaboration of the ASM modeling and refinement concepts is given by Börger and Raschke [7].

2.3 ASM Language Representation

In this study, we use the basic syntax elements from the CASM language[5] [52]. The CASM language elements used can be found in a similar fashion in other ASM languages; hence, we believe it is likely that our results can be applied to other ASM languages. CASM is a statically typed ASM-based specification language. Every specification is composed of definition elements. Relevant to this study are the following three definitions – *Function*, *Derived*, and *Rule* definitions.

Function Definition. A `function` definition specifies an n-dimensional state (argument types) which maps to a certain function type (return type). E.g. variables in a programming language are modeled as nullary *functions* in ASMs, or hash-maps can be expressed as unary *functions* in ASMs. Listing 1 illustrates the concrete syntax and some examples.

Derived Definition. A `derived` definition specifies functions which state values can only be derived from other *functions* or *deriveds* without modifying the ASM *state*. Therefore, *derived* functions are side-effect free. Listing 2 illustrates the concrete syntax and some examples which use state information from Listing 1.

[5] See https://casm-lang.org/syntax for CASM language description.

Rule Definition. A `rule` definition specifies a *named rule* (language user defined *rule*) which describes the actual computation and transaction of the ASM state evolving expressed through basic ASM *rules* namely: (1) *update* rule to produce a new value for a given state function (*location*); (2) *block* rule to express bounded parallelism of multiple *rules*; (3) *sequential* rule to express sequential execution semantics of multiple *rules*; (4) *conditional* rule to specify branching (`if-then-else`); (5) *forall* rule to express parallel computations; (6) *choose* rule to specify nondeterministic choice; (7) *iterate* rule to express iterations; and (8) *call* rule to invoke *named rules* (sub-rule call). A more detailed explanation of all ASM *rules* is given by Börger and Raschke [7]. Listing 3 illustrates the concrete syntax and an example which depends on some definitions from Listing 1 and Listing 2.

2.4 Experiment Language Construct Representations

Besides a class concept used in AsmL [26], no other object-oriented language construct has been introduced in the ASM language and tool landscape. To enable moving the state-of-the-art in advanced object-oriented abstractions for such formal languages forward, this study tests two language construct representations, namely interfaces and traits, to search for a suitable object-oriented abstraction to structure state and behavioral aspects for such languages in general and specifically for CASM. In order to do so, we introduced three new definitions for this study into the existing CASM syntax – *Feature*, *Structure*, and *Implement* definitions.

Feature Definition. A `feature` definition specifies a new type (functionality) together with a set of operations (*derived* and *rule* declarations) which form a *protocol*.

Structure Definition. A `structure` definition specifies a composition of (function) states which can be extended with one or multiple *features* (functionalities).

Implement Definition. An `implement` definition specifies which *feature* gets implemented and used by which *structure*. This definition element binds default or extended functionalities (behaviors) to a certain type (structure).

Please note that we use these very general terms on purpose as they can be mapped to the two language constructs under investigation. As a consequence, we can avoid bias from participants in the experiment are who know keywords identifying the language construct through `interface` or `trait` which especially applies for the keyword `feature`. The syntax of the two language constructs are designed in the style of modern object-oriented programming languages.

Language Construct Interfaces (Experiment Group A). The `feature` syntax in the language construct *Interfaces* only describes the *protocol* consisting of the set of operations [45] [9] a `structure` has to implement. Therefore, it consists only of `derived` and/or `rule` declarations. In order to use a `feature`, the keyword `implement` has to be used to extend the current `structure`. Listing 4 depicts an example specification with the *Interface* language construct[6]. This syntax is primarily influenced by the Java programming language [58] interface syntax.

[6]See `form_ifaces.pdf` at [54].

```
feature Formatting = {
  derived toString : -> String
}

structure Person implement Formatting = {
  function name : -> String
  function age : -> Integer



  derived getName -> String = this.name
  derived getAge -> Integer = this.age

  rule setName( name : String
) = this.name := name
  rule setAge( age : Integer ) = this.age := age



  // encapusalted feature implementation
  derived toString -> String =
    this.getName() + ( this.getAge() as String )
}
```

Listing 4. *Interfaces*-Based Example Specification

```
feature Formatting = {
  derived toString -> String
}

structure Person = {
  function name : -> String
  function age  : -> Integer
}

implement Person = {
  derived getName -> String = this.name
  derived getAge -> Integer = this.age

  rule setName( name : String
) = this.name := name
  rule setAge( age : Integer ) = this.age := age
}

// decoupled feature implementation
implement Formatting for Person = {
  derived toString -> String =
    this.getName() + ( this.getAge() as String )
}
```

Listing 5. *Traits*-Based Example Specification

Language Construct Traits (Experiment Group B). The `feature` syntax in the language construct *Traits* is equal to *Interfaces* except that it supports definition of optional default implementations inside the `feature` definition itself. A `structure` only contains the state information. The behavior in the *Traits* abstraction is implemented through two different kinds of separated `implement` definitions: (1) describes the behavior of the structure; (2) describes the behavior of a certain `feature for` a structure. It is important to note here that a default implementation provided in the `feature` syntax can be overwritten in the `implement` definition. Listing 5 depicts an example specification with the *Traits* language construct[7]. This `feature` and `implement` syntax is influenced by the Rust programming language [47] trait syntax[8].

2.5 Related Studies

So far, interfaces and traits have mainly been studied in the context of programming languages and mainly by proposing new solutions. A small number of empirical studies exists in this field which are mainly case studies. For instance, Murphy-Hill et al. present a case study on the potential of traits to reduce code duplication [48]. However, so far no study comparing the two language constructs interfaces and traits covered in our study exists and also no controlled experiments.

Interface abstractions have been extensively studied in the context of formal methods [13] [17] [11] and architecture description languages that offer formal representations [50] [21]. Traits in contrast have not yet been studied in the context of formal methods. We are not aware of any formal method that unifies or integrates the two object-oriented language constructs covered in our study.

Overall formal methods have been studied before in only a few empirical studies other than case studies. An example of the few existing studies is the one by Sobel and Clarkson, who study the aiding effect of first-order logic formalisms in software development [71]. Czepa and Zdun [16] and Czepa et al. [15] have studied the understandability of formal methods for temporal property specification using similar research methods as used in this study.

[7] See `form_traits.pdf` at [54].
[8] See https://doc.rust-lang.org/rust-by-example/trait.html for Rust's trait syntax description.

Snook and Harrison [70] performed structured interviews with formal method users asking them about scalability, understandability, and tool support issues. A very interesting aspect of this study is that the participants report that *"the precise and accurate nature of the specification makes the coding task straightforward and the coder is less likely to build in redundant code."* [70]. Another interesting finding in this study is that the *"interviewees thought that the difficulties with using formal specifications were in finding the useful abstractions from which to create models."* [70]. Snook and Harrison [70] argue that the problem behind the interviewees statement is that programming languages mainly focus on structural aspects first whereas formal methods focus on behavioral aspects.

We are not aware of any empirical study systematically investigating object-oriented language constructs in the context of state-based formal methods. Only, in our own prior work we conducted a study [55] with 105 participants where we analyzed how well experiment participants understand given ASM specifications which are represented in three different language constructs – interfaces, mixins, and traits. The results of this experiment showed that the object-oriented abstractions interfaces and traits are better understandable than mixins.

3 EXPERIMENT PLANNING

This study is structured following the guidelines by Jedlitschka et al. [33] on how empirical research shall be conducted and reported in software engineering. Moreover, the guidelines by Kitchenham et al. [36], Wohlin et al. [75], and Juristo and Moreno [34] for empirical research in software engineering were used in our study design. For the statistical evaluation of the acquired data we considered and applied the *robust statistical method* guidelines for empirical software engineering by Kitchenham et al. [35].

3.1 Goals

The **goal of this experiment** is to **measure the construct specification effectiveness** and **efficiency** on how well and fast a participant understands a given problem provided as informal textual description and expresses an ASM specification as textual representation using one **of the two different language constructs**, namely *Interfaces* and *Traits*. The quality focus of the construct *specification effectiveness* and *efficiency* is the *correctness* and *duration* of the participant's modeled ASM specification solution.

3.2 Context and Design

This study reports on a **controlled experiment with** 98 **participants** in total to study the specification effectiveness and efficiency of the language constructs interfaces and traits in the context of ASMs. We used a **completely randomized design** with one alternative per experimental group, which is appropriate for the stated goal. Through this, we tried to avoid learning effects of the participants and experimenter bias in the assignment of the groups. The statistical evaluation technique is based on measuring how well a participant understands a given problem by specifying an appropriate solution written as textual representation in an ASM language.

3.3 Participants

All 98 participants of the experiment are Bachelor of Science (BSc) students of the Faculty of Computer Science at the University of Vienna, Austria enrolled in the course Software Engineering 2 (SE2)[9] in the winter term 2018/19. The BSc students enrolled in the SE2 course are used as proxies for novice to moderately advanced software architects, designers, or developers. This course, which is a mandatory part of the BSc curricula at the University of Vienna, is intended for students in the fourth semester of the BSc curricula. The content of this course is about teaching principles of the construction and design of software systems, investigating different methods and tools, design patterns, programming styles, and how to tackle non-functional requirements. The participants (students) received training in programming, software engineering, (data) modeling, basic formal methods, algorithms, and mathematics in previous courses.

At the beginning of the SE2 course, the students were informed that during the semester there will be an opportunity to participate in an experiment. The attendance of the experiment was optional, and the submitted solutions (filled out survey forms) were rewarded with up to 6 bonus points. There was the option to receive the 6 bonus points by performing the tasks, but not participate in the experiment (opt out option). How well (correctness, see Section 5.1) a participant answered the survey determined the bonus points. In total, there were 98 participants, which were randomly allocated to the treatments (using one of the two language construct representations in an ASM specification language, see Section 2). Due to random assignment of the participants to groups – *Interfaces* (Group A) and *Traits* (Group B) – the final distribution resulted in 49 : 49. Some may argue that students as experiment participants are not good proxies for novice software engineers. The experiment participants are students of an advanced course (SE2) at the University of Vienna, which trained the students in abstractions needed for the experiment task domain, and were trained in basic formal methods in prior courses. Easy to understand formalisms are key to correct specifications in practice. We expect advanced students to be good proxies for inexperienced developers and architects.

In this study, we do not focus on well trained experts as they are usually also much better trained in formalisms, because the goal of the study is not to focus on techniques that can only be applied by a few very well trained experts. Furthermore, according to Kitchenham et al. [36] using students *"is not a major issue as long as you are interested in evaluating the use of a technique by novice or nonexpert software engineers. Students are the next generation of software professionals and, so, are relatively close to the population of interest"*. This is directly reflected in this study because some of the students who participated in the experiment show several years of programming experience as well as several years of work experience in the software and/or hardware industry (see Figure 2d). Other studies by Svahnberg et al. [73] or Salman et al. [62] would argue even further and state that under certain circumstances, students are valid representatives for professionals in empirical software engineering experiments.

3.4 Material and Tasks

The experiment is based on a selection of basic software system applications. The selection includes a *Calculator System*, an *Event Scheduling/Pooling System*, and a *Traffic Control System* as example applications inspired by some examples provided by Börger and Raschke [7].

[9] See https://ufind.univie.ac.at/en/course.html?lv=051050&semester=2018W for SE2.

The *Calculator System* example focuses on the aspect on the decomposition of states and behaviors of a client-server application by defining and reusing a message-based interface or trait between them.

In the *Event Scheduling/Pooling System* example a participant shall express the use of abstract behavior by using interface-based or trait-based parameters (behavioral typed parameters) to separate the event scheduling from the event execution behavior.

The *Traffic Control System* example focuses expressing, mixing, and reusing multiple behaviors to form and compose certain structural state properties. Therefore, the key aspect in this example application is to detect which behavior can be expressed through a proper interface or trait and can be combined to achieve certain structural state property.

The principles and concepts to comprehend the given example system applications are related to the subjects taught in the SE2 course. This study consists of two major experiment material artifacts:

(1) Information Sheet An experiment information document[10] explaining the ASM language syntax and semantics **without the experiments' language construct** syntax and semantics extensions.

(2) Survey Form Two experiment survey forms[11] per experimental group and language construct containing the actual survey along **with the explicit experiments' language construct** syntax and semantics extension and description **per experimental group**.

The two experiment *survey forms* are structured the same way consisting of four parts: (1) a participant background information questionnaire; (2) the experimental group language construct syntax and semantics extension description; (3) three experiment tasks (equal to all experiment groups); and (4) an overall experiment questionnaire at the end. Each experiment task is divided into three sections:

(1) Informal Description of a selected software system application as an informal textual representation. The students (participants) were instructed to read and understand the given informally described software system application **before** they start to process the next section of the experiment task.

(2) Formal Specification is an open question field where the participants were instructed to write down the corresponding ASM specification for the given informally described software system application by using the experimental group assigned language construct syntax extension for the ASM language.

(3) Self Assessment is a questionnaire used to obtain a perspective of the participants' self assessment of how correct their answers are with a certain level of confidence.

Important is that all task sections are identical for both experiment groups, since only in the participants' written solution a difference is visible due to the different assigned treatment (language construct) in the modeled ASM specification.

3.5 Variables and Hypotheses

The independent variables (factors) for this controlled experiment have two treatments, namely the two different representations of the language constructs *Interfaces* and *Traits*. The dependent variables of this study are measured through:

[10] See `info.pdf` at [54].
[11] See `form_ifaces.pdf` and `form_traits.pdf` at [54].

(1) Correctness The specification effectiveness (correctness) is derived from the participants' modeled ASM specification and examined through evaluation criteria by analyzing structural, behavioral, reusable, functional, and syntax properties.

The precise description on how the correctness is computed is given in Section 5.1.

(2) Duration The specification efficiency (duration) is the time it took the participants to comprehend the informal specification (stimuli) and model a corresponding ASM specification by using one of the two object-oriented abstractions. Important to note here is that the measurement of the duration variable only includes the processing time (reading, comprehending, and writing) and excludes breaks (see Section 3.4).

We hypothesized that *Traits* are easier to comprehend than *Interfaces* due to the fact that *Traits* have the ability to avoid code duplication and clearer separation of state and behavioral aspects by having almost equal Application Programming Interface (API) declaration styles as *Interfaces*. Consequently, as suggested by Wohlin et al. [75] we formulate the following null hypotheses, where specification effectiveness is measured by the correctness variable and specification efficiency is measured by the duration variable:

$H_{0,1}$ The specification effectiveness shows no significant difference (similar performance) for *Interfaces* compared to *Traits*.

$H_{0,2}$ The specification efficiency shows no significant difference (similar performance) for *Interfaces* compared to *Traits*.

From the null hypotheses above we can derived and formulate the following alternative hypotheses, for this controlled experiment:

$H_{A,1}$ The specification effectiveness shows a significant difference (better performance) for *Traits* compared to *Interfaces*.

$H_{A,2}$ The specification efficiency shows a significant difference (better performance) for *Traits* compared to *Interfaces*.

4 EXPERIMENT EXECUTION

This experiment was executed in two steps – a preparation and a procedure phase.

4.1 Preparation

Two weeks before the experiment we handed out the preparation material (the experiment *information sheet*, see Section 3.4) through an e-learning platform[12]. This document provided general information of the upcoming experiment and an introduction to the ASM language syntax and semantics used without explaining one of the two language constructs. All ASM language concepts used are depicted with short example ASM specification snippets. The participants were allowed to use this document during the experiment in printed form. The main reason why we provided the experiment information document is that all participants needed to be educated to the same level of detail with regard to a state-based formal method and specifically to a concrete ASM language representation (see Section 2).

[12]See https://moodle.org for e-learning platform information.

4.2 Procedure

The experiment was carried out using paper and pencil, as if it were an (closed book) exam. Participants were allowed to bring only one aid – the *information sheet* – to process the experiment survey form as described in the previous Section 4.1. At the beginning of the experiment, every participant received a random experiment *survey form* (see Section 3.4). They were instructed to fill out and process the survey from the first page to the last page in this particular order. Furthermore, a clock with seconds granularity was projected onto a wall to provide timestamp information to the participants. They were asked to track start and stop timestamps during the processing of the experiment tasks. After the experiment every participants' modeled ASM specification was examined through a list of evaluation criteria (see Section 5.1) and the results of the examination was recorded in a spreadsheet. The participants' task start and stop timestamps were converted to a duration in seconds and summed up to a total duration for all tasks. We used the four-eyes principle during every manual work step (answer obtaining and timestamp conversion) in the data collection. The experiment execution and data collection were performed as described in this Section and we have not observed any form of deviations or unforeseen difficulties.

5 ANALYSIS

All statistical analysis was performed with the software tool R[13]. The analysis processes[14] contain the following steps: (1) load the prepared data-set from Section 5.1; (2) calculate the descriptive statistics for the dependent variables which are explained in detail in Section 5.2; (3) perform a group-by-group comparison with appropriate statistical hypotheses tests which are explained in detail in Section 5.3; (4) generate table/plot information in order to include this information in this article. In order to reproduce the analysis results, some R library package dependencies have to be installed[15].

5.1 Data-Set Preparation

The raw data[16] collected during the experiment execution phase (see Section 4) was prepared[17] in the following manner: (1) the obtained LibreOffice OpenDocument Spreadsheet (ODS) file [51] was exported to a Comma-Seperated Values (CSV) file [67]; (2) the CSV file was imported for further processing; (3) type castings of several data rows were performed; (4) the calculation of task-based and overall **Duration** times; (5) the calculation of task-based and overall **Correctness** values; and (6) stored as an R Data-Set (RDS) file [59] for further processing and analysis.

The calculation of the **Correctness** value is composed out of a check list of yes-and-no statements[18] for all the different tasks in the experiment survey forms (see Section 3.4). This list of yes-and-no statements was derived before the experiment execution by specifying ground truth models for both object-oriented language abstractions variants – interfaces and traits – of the informal described experiments' example software application systems. In order to enable a flexible way to compare the participants' solutions from the

[13] See https://www.r-project.org for version `3.5.2`.
[14] See `analyze.r` at [54].
[15] See `install.r` at [54].
[16] In order to enable reproducability of our results, the data-set (`README.ods`) is made public in the long term open data archive Zenodo [54] together with all documents and R scripts.
[17] See `prepare.r` at [54].
[18] See `README.ods` for the complete list of the yes-and-no statements along with the collected data for all participants at [54].

Table 1. Number of Yes-and-No Statements per Evaluation Criteria and Tasks

Evaluation Criteria	Task 1	Task 2	Task 3	All Tasks
Structure	4	3	5	12
Behavior	4	3	5	12
Syntax	5	5	7	17
Reusability	4	4	6	14
Functionality	4	2	4	10
Total	21	17	27	65

experiment, the obtained list of yes-and-no statements reflects generic properties the provided and specified models by the participants shall contain. The yes-and-no statements are grouped into five evaluation criteria (categories) – structure, behavior, syntax, reusability, and functionality. The following list depicts for each of the evaluation criteria an example yes-and-no statement:

(1) Structure Did the participant specify certain structural elements? An example structural evaluation criteria statement for Task 1[19] is defined as follows: *"Proxy structure defined"*?

(2) Behavior Did the participant specify certain behavioral elements? An example behavioral evaluation criteria statement for Task 1[19] is defined as follows: *"Client implemented default behavior"*?

(3) Syntax Did the participant use the correct language construct syntax for the assigned treatment? An example syntactical evaluation criteria statement for Task 1[19] is defined as follows: *"Server valid abstraction syntax"*?

(4) Reusability Did the participant recognized reusable elements and did (s)he specify it through the correct language construct syntax for the assigned treatment? An example reusable evaluation criteria statement for Task 1[19] is defined as follows: *"Operations implemented for Proxy"*?

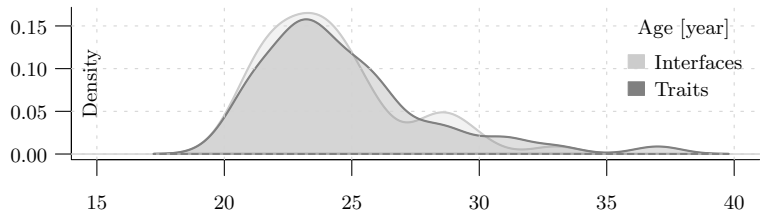
(5) Functionality Did the participant specify certain functionalities? An example functional evaluation criteria statement for Task 1[19] is defined as follows: *"Message provides unique identification"*?

In total there exist 65 yes-and-no statements per experiment participant. By accumulating the percentage value of all yes-and-no statements a total of 100% **correctness**[20] can be achieved. Table 1 depicts the number of yes-and-no statements in total and the dissection per evaluation criteria and tasks.

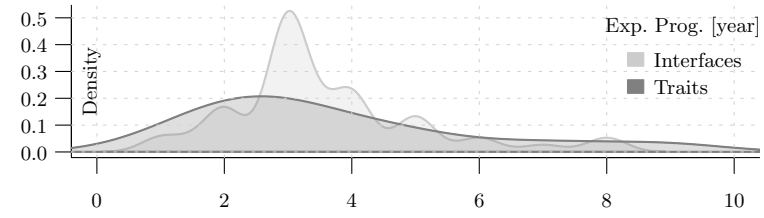
5.2 Descriptive Statistics

Background Information. The participants' experience and characteristics are captured in the experiment through eight parameters[21] and the results indicate that overall, the random distribution of the participants to the experiment groups is almost balanced. The participants' age (see Figure 2a) shows a similar distribution for both groups with a peak around 23 years. The programming experience of the participants measured in years (see Figure 2b) indicate that the interfaces group has a more than twice higher density around 3 years of experience in programming compared to the traits group which has its peak around 2.5. This is the only background information parameter showing a slightly unbalanced distribution and indicates that the general programming experience level is higher in the interfaces experiment group. This discrepancy is attributed to the randomized distribution of the experiment survey to the participants.

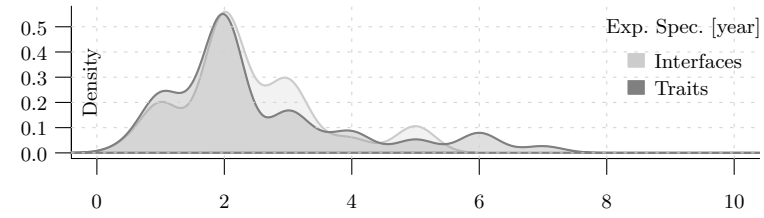
[19] See `form_ifaces.pdf` or `form_traits.pdf` for description of Task 1 at [54].
[20] For detailed formula, see `prepare.r` Line 97-250 at [54].
[21] See `appendix.pdf` at [54] for more detailed supplementary background information.



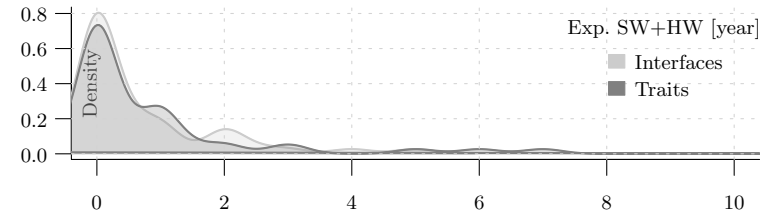
(a) Kernel Density Plot of Participants' Age



(b) Kernel Density Plot of Participants' Programming Experience in Years



(c) Kernel Density Plot of Participants' Specifying Experience in Years



(d) Kernel Density Plot of Participants' SW/HW Industry Experience in Years

Fig. 2. Descriptive Plots per Group of Participants' Background Information

Table 2. Participants' Gender

Gender	Interfaces	Traits
Female	20	17
Male	29	32

Table 3. Participants' Level of Education

Education	Interfaces	Traits
None	42	45
BSc	7	4

Table 4. Participants' Programming Language Knowledge

Language	Interfaces	Traits
Java	49	49
Cpp	46	48
PHP	41	39
C	13	17
Scala	11	16
Swift	7	3
Assembler	3	5
Basic	2	3
Fortran	2	2
Rust	1	0
Kotlin	0	3
Haskell	0	2

Table 5. Participants' Prior Knowledge of Formal Methods

Interfaces	Traits
5	4

In contrast to the programming experience, the distribution of the participants' specification (modeling) experience measured by years (see Figure 2c) is quite similar for both groups with a peak at 2 years. Since our participants are students, the peak of the software (SW) and hardware (HW) industry experience measured in years (see Figure 2d) is at zero years, but a number of students show a similar level of industry experience between 1 to 3 years.

The experiment total ratio between female and male participants is 37 (37.76%) : 61 (62.24%). The interfaces group has 20 (40.82%) female and 29 (59.18%) male participants and the traits groups has 17 (34.69%) female and 32 (65.31%) male participants.

From the perspective of prior computer science education (see Table 3) only 11 (11.22%) students have a previous BSc degree and the other 87 (88.78%) participants are undergraduates. The numbers are quite comparable in the two experiment groups. All participants (100%) are familiar with Java and 94 (95.92%) participants – 46 (93.88%) interfaces group and 48 (97.96%) traits group – are familiar with C++. That means the interface abstraction should be more than familiar to both experimental groups. We can further observe languages offering traits, besides the programming language PHP (total 80 (81.63%) – interfaces group 41 (83.67%) and traits group 39 (79.59%)), are rather underrepresented in both experimental groups. This is the case for the programming languages Scala (total 27 (27.55%) – interfaces group 11 (22.45%) and traits group 16 (32.65%)), Swift[22] (total 10 (10.20%) – interfaces group 7 (14.29%) and traits group 3 (6.12%)), and Rust where only one of all participants (interfaces group 2.04%) is familiar with the language.

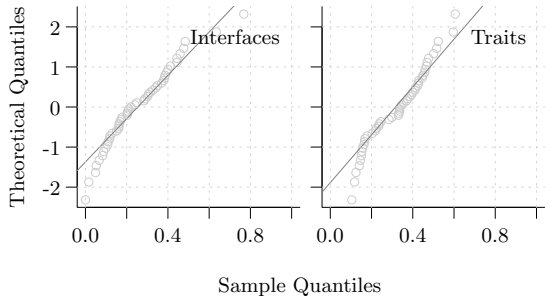
A very important parameter of the background information is if there are participants which have a prior knowledge of formal methods (see Table 5). Accordingly to the obtained results, only 9 participants (9.18%) in total – interfaces group 5 (10.20%) and traits group 4 (8.16%) – have stated that they have prior knowledge in a formal method.

Dependent Variable **Correctness**. Table 6 contains the number of observations, central tendency measures, and dispersion measures per language construct for the dependent variable **Correctness**[23] and this acquired data is visualized as a kernel density plot in Figure 3b and a box plot in Figure 3c. In the box plot we can observe that the median of the *Interfaces* group is almost at the lower quartile value of the *Traits* group. There is one outlier in the *Interfaces* group which performed very well.

The distribution of the *Interfaces* group is left skewed whereas the *Traits* group is right skewed. The *Traits* group has no outlier at all. According to the kernel density plot, the data does not appear to be normally distributed, and both distributions look different, which implies unequal variances and both distributions have two peaks as well. The *Interfaces* group has one peak at 0.16 and another one at 0.37 whereas the *Traits* group has one peak at 0.17 and another one at 0.41.

Dependent Variable **Duration**. Table 8 contains the number of observations, central tendency measures, and dispersion measures per language construct for the dependent variable **Duration**[24] and this acquired data is visualized as a kernel density plot in Figure 4b and a box plot in Figure 4c. In the box plot we can observe that for both groups the median is almost the same (*Interfaces* at 3935 and *Traits* at 3980), but the lower and upper quantiles of the *Traits* group indicate a wider distribution which is reflected in Figure 4b. The latter shows the data does not appear to be normally distributed for the *Interfaces* group and almost for the *Traits* group, and the two distributions look different, which implies unequal variances. The *Interfaces* group has its peak at 3950 seconds and the *Traits* group has its peak at 4000 seconds. Moreover, the box plot shows three outliers for the *Interfaces* group – two participants which processed the experiment (*survey form*) really fast and one participant who processed it really slow.

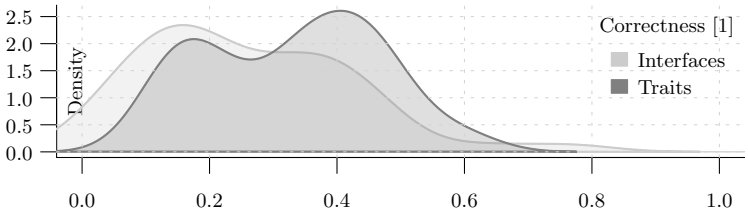
[22]Swift has implemented traits through the *protocol extension* syntax. See, e.g. https://docs.swift.org/swift-book/LanguageGuide/Extensions.html.
[23]Unit is correctness rate between 0.0 and 1.0 (denoted [1]).
[24]Unit is duration in seconds (denoted [s]).



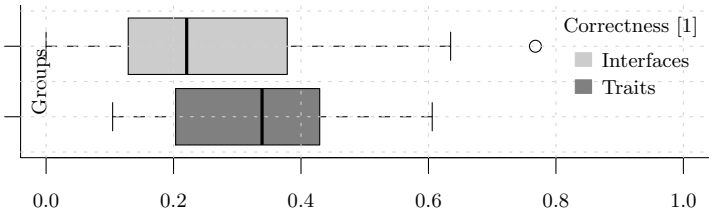
(a) Normal Q-Q Plot of **Correctness**

Table 6. Descriptive Statistics per Group of Dependent Variable **Correctness**

	Interfaces	Traits
Number of observations [1]	49	49
Mean [1]	0.2585	0.3283
Standard deviation [1]	0.1624	0.1370
Median [1]	0.2206	0.3389
Median abs. deviation [1]	0.1673	0.1737
Minimum [1]	0.0000	0.1044
Maximum [1]	0.7678	0.6059
Skew [1]	0.7353	0.0061
Kurtosis [1]	0.4169	-1.1433
Shapiro-Wilk Test p [1]	0.0437	0.0421



(b) Kernel Density Plot of **Correctness**



(c) Box Plot of **Correctness**

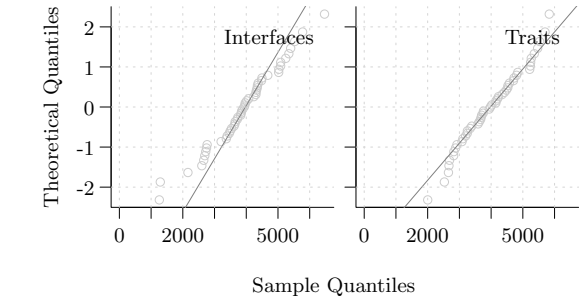
Table 7. Hypothesis Tests per Group Combination of the Dependent Variable **Correctness**

	Interfaces vs. Traits
Cliff's δ	0.2932
s_δ	0.1109
v_δ	0.0123
z_δ	2.6449
CI_{low}	0.0635
CI_{high}	0.4934
$P(X > Y)$	0.3528
$P(X = Y)$	0.0012
$P(X < Y)$	0.6460
p	0.0095
p_{FDR}	**0.0191**
Effect Size	small

Fig. 3. Descriptive Plots per Group of the Dependent Variable **Correctness**

5.3 Hypothesis Testing

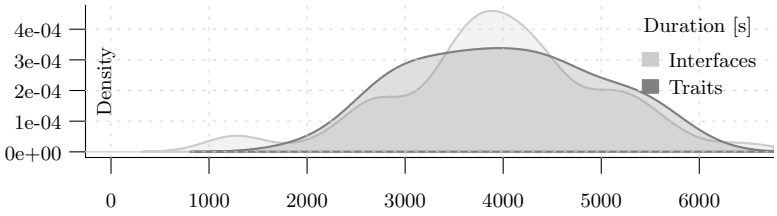
Due to the presence of two experiment groups and two dependent variables, the Multivariate Analysis of Variance (MANOVA) [6] would be a suitable statistical procedure, but necessary assumptions must be met to apply this method. The investigation of the kernel density plots – Figure 3b for **Correctness** and Figure 4b for **Duration** – indicates that not all distributions of the experiment groups are normally distributed, which the MANOVA would need in order to be applied. We applied the Shapiro-Wilk normality test [68] (last row in Table 6 and Table 8) and for both groups (*Interfaces* and *Traits*) for the dependent variable **Correctness** shows a significant ($p \leq 0.05$) difference to the normal distribution, which would make MANOVA not suitable for **Correctness** but suitable for **Duration**. To finally conclude that the MANOVA method cannot be applied, we visually inspected the normal Q-Q plots for both dependent variables, which are depicted in Figure 3a for **Correctness** and Figure 4a for **Duration**. All distribution plots indicate



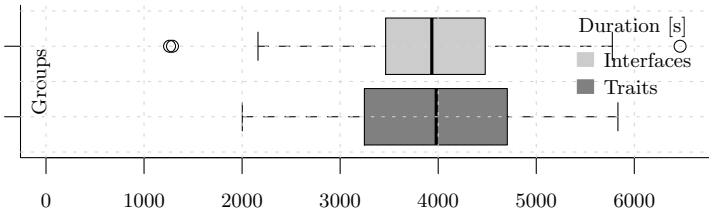
(a) Normal Q-Q Plot of **Duration**

Table 8. Descriptive Statistics per Group of Dependent Variable **Duration**

	Interfaces	Traits
Number of observations [1]	49	49
Mean [s]	3937.96	3997.45
Standard deviation [s]	1060.92	960.31
Median [s]	3935.00	3980.00
Median abs. deviation [s]	794.67	1086.75
Minimum [s]	1260.00	2002.00
Maximum [s]	6467.00	5833.00
Skew [1]	-0.2517	0.0730
Kurtosis [1]	0.2615	-0.9831
Shapiro-Wilk Test p [1]	0.4969	0.4108



(b) Kernel Density Plot of **Duration**



(c) Box Plot of **Duration**

Table 9. Hypothesis Tests per Group Combination of the Dependent Variable **Duration**

	Interfaces vs. Traits
Cliff's δ	0.0217
s_δ	0.1179
v_δ	0.0139
z_δ	0.1837
CI_{low}	-0.2074
CI_{high}	0.2484
$P(X > Y)$	0.4890
$P(X = Y)$	0.0004
$P(X < Y)$	0.5106
p	0.8547
p_{FDR}	0.8546
Effect Size	negligible

Fig. 4. Descriptive Plots per Group of the Dependent Variable **Duration**

that the linearity assumption is not met and the power of the test might be affected. Thus we ruled out multivariate and parametric testing because it could lead to unreliable results.

Instead, we selected a non-parametric testing method. When we considered our acquired data, according to Kitchenham et al. [35], we cannot use the Kruskal-Wallis test [38] because it is strongly affected by unequal variances. Therefore, we select a robust non-parametric test called Cliff's δ [14]. This testing method is unaffected by non-normal data, change in distribution, and (possible) unstable variance.

The results of the Cliff's δ test is shown in Table 7 for the dependent variable **Correctness** and in Table 9 for the dependent variable **Duration**. Due to the fact that we applied this hypothesis test two times, we are required to lower the significance level in order to avoid Type I errors, which is about not detecting an effect that is not present. A suitable approach would be to apply the Bonferroni correction [18], which suggests to lower the current significance level $\alpha = 0.05$ divided by the times a certain test was applied

($n = 2$), which would result into $\alpha' = \frac{\alpha}{n} = \frac{0.05}{2} = 0.025$. Unfortunately, this significance level correction is known to increase Type II errors, which is about not detecting an effect that is present. Therefore, we choose a more robust correction method which does not increase Type II errors, namely the False Discovery Rate (FDR) adjusted p-values [4]. According to the FDR adjusted p-values (p_{FDR}) in Table 7 and Table 9, there is evidence to reject one of the hypotheses of this study (see Section 3.5). For the dependent variable **Correctness** we found evidence of a better specification effectiveness of expressing structural, behavioral, syntactical, reusable, and functional aspects through ASM specifications from a given informal description of software system applications. The test results on **Correctness** are significant with a small effect size magnitude [35] for the comparison of *Interfaces* and *Traits*, which suggests to reject $\mathbf{H_{0,1}}$ and to accept $\mathbf{H_{A,1}}$. For the dependent variable **Duration** the null hypothesis $\mathbf{H_{0,2}}$ cannot be rejected as the test results are not significant. Therefore, the alternative hypothesis $\mathbf{H_{A,2}}$ cannot be accepted.

6 DISCUSSION

This section covers the evaluation, implications, threats to validity, inferences, and relevance to practice.

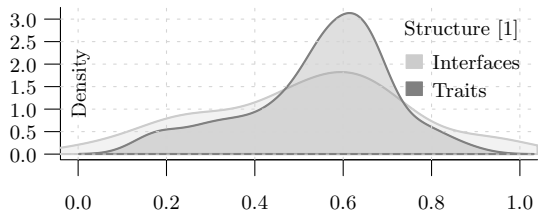
6.1 Evaluation of Results and Implications

The descriptive statistics do directly favor one of the language constructs, because by looking at the dependent variable **Correctness**, *Traits* performs better than *Interfaces*. The median of the **Correctness** variable is for language construct *Interfaces* 22.06% and *Traits* 33.89%. Due to the fact that all participants have almost no prior knowledge (< 10%) of ASMs and formal methods in general (checked by an informational question in the survey, see Section 5.2), a median for the specification effectiveness (correctness) between 22% to 34% can be considered a rather good result in this study. For the **Duration** descriptive statistical results, *Interfaces* and *Traits* seem to have a similar distribution. The median of the **Duration** variable is for language construct *Interfaces* 3935s (1h 5min 35s) and *Traits* 3980s (1h 6min 20s), which are good results in the scope of the processed survey and the achieved **Correctness** results with a limited experiment time of 120min (2h). Note that the highest participant duration was 6467s (1h 47min 47s).

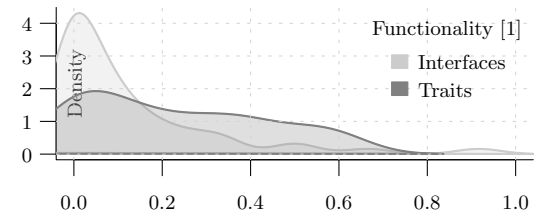
In the inferential statistics *Traits* show a significantly better performance than *Interfaces* in terms of **Correctness** (specification effectiveness). This significance implies that for the ASM language user (novice software developer or designer) it is easier and more effective to express informal descriptions and their properties with *Trait*-based ASM specifications rather than with *Interface*-based ASM specifications.

In order to explain and gain more details about the better **Correctness** results for the *Traits* group compared to the *Interfaces* group, we have dissected the correctness to the five evaluation criteria (see Section 5.1) and analyzed them individually.

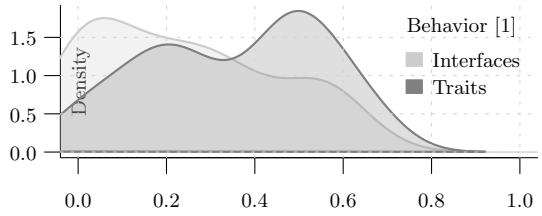
The structural correctness (see Figure 5a) value shows a density about twice as high for the *Traits* group with a peak correctness value for both groups around 61%. The distribution of the behavioral correctness (see Figure 5b) depicts that the participants of the *Traits* group performed much better (peak around 50%) in specifying behavioral aspects in the provided ASM specification solution compared to the *Interfaces* group (peak around 7.5%). It is interesting that the results on the reusability properties (see Figure 5c) of the specified ASM specifications performed only slightly better for the *Traits* group. This indicates, together with the low correctness values, that the participants had problems to detect possible interfaces inside the informal descriptions of the software system applications.



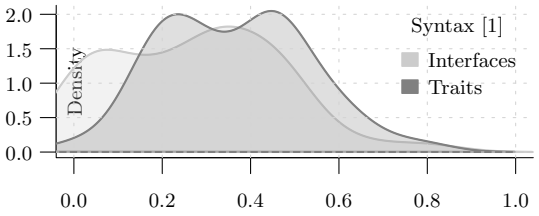
(a) Kernel Density Plot of Structural Correctness



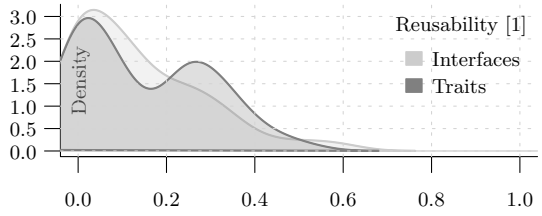
(d) Kernel Density Plot of Functionality Correctness



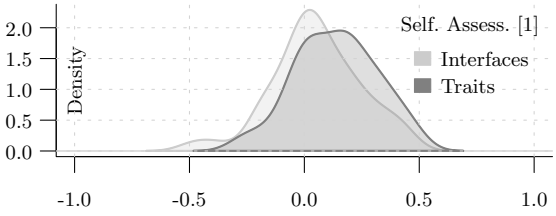
(b) Kernel Density Plot of Behavioral Correctness



(e) Kernel Density Plot of Syntax Correctness



(c) Kernel Density Plot of Reusability Correctness



(f) Kernel Density Plot of Participants' *Self Assessment*

Fig. 5. Descriptive Plots per Group of Correctness Evaluation Criteria and Participants' *Self Assessment*

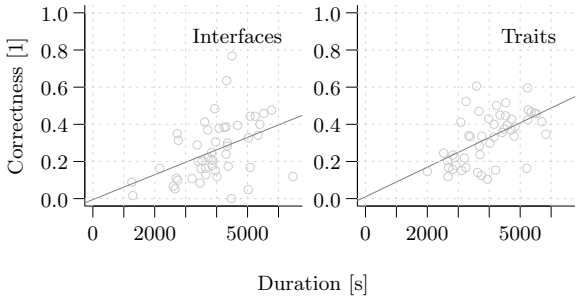


Table 10. Correlation per Group of the Dependent Variables **Correctness** to **Duration**

	Interfaces	Traits
Spearman's ρ	0.4980	0.5596
Pearson's r	0.4374	0.5584

Fig. 6. Scatter Plot per Group of the Dependent Variables **Correctness** to **Duration**

The distributions of the functionality correctness (Figure 5d) show that a large number of participants of the *Interfaces* group were not able to express functionalities very well. The *Traits* group, in contrast, shows a very stretched distribution from 0% up to 65%. Apparently the participants were able to express (non object-oriented related) functionalities better through the *Traits*-based ASM syntax extension. Figure 5e

Table 11. Questionnaire Results $\mathbf{Q_n}$

(a) Results of $\mathbf{Q_1}$ (Stimuli)

Q_1	Interfaces	Traits
strongly agree	4	4
agree	20	19
neutral	13	14
disagree	10	8
strongly disagree	2	4

(b) Results of $\mathbf{Q_2}$ (Structural)

Q_2	Interfaces	Traits
strongly agree	2	4
agree	17	10
neutral	11	17
disagree	18	10
strongly disagree	1	8

(c) Results of $\mathbf{Q_3}$ (Behavioral)

Q_3	Interfaces	Traits
strongly agree	1	2
agree	9	5
neutral	11	7
disagree	21	22
strongly disagree	7	13

(d) Results of $\mathbf{Q_4}$ (Functionality)

Q_4	Interfaces	Traits
strongly agree	1	1
agree	8	2
neutral	10	16
disagree	23	15
strongly disagree	7	15

(e) Results of $\mathbf{Q_5}$ (Interfaces)

Q_5	Interfaces	Traits
strongly agree	15	15
agree	22	24
neutral	6	6
disagree	5	1
strongly disagree	1	3

(f) Results of $\mathbf{Q_6}$ (Traits)

Q_6	Interfaces	Traits
strongly agree	2	1
agree	7	5
neutral	3	5
disagree	21	16
strongly disagree	16	22

compares syntactical correctness results. We can observe that both groups' distribution have two peaks – 7% and 35% for the *Interfaces* group, and 21% and 45% for the *Traits* group.

The kernel density plot for the participants' *self assessment* is depicted in Figure 5f. The self assessment was measured by calculating the difference between the actual **Correctness** value and the participants **Confidence** value that a certain solution to a task they worked on was correct. A self assessment value ≤ 0 means the participant overestimated and ≥ 0 means the participant underestimated the **Correctness** of the given experiment answers. Both experiment groups show almost a similar *self assessment* with its peak in the underestimated section. This implies that both object-oriented abstractions show a similar participants' self assessment regarding their **Confidence** in the **Correctness** of their given solutions.

Studying the scatter plot (Figure 6), Spearman's rank correlation, and Pearson product-moment correlation (Table 10) of the two dependent variables **Correctness** and **Duration**, we cannot observe a clear (linear nor a non-linear) monotonic trend that the dependent variables are strongly correlated somehow.

As described in Section 3.4 we also asked the participants to fill in a post experiment questionnaire where they could provide us answers using six Lickert-scale [44] questions ($\mathbf{Q_n}$) with five possible answers: (1) strongly agree, (2) agree, (3) neutral, (4) disagree, (5) strongly disagree. The questions and their corresponding results are:

$\mathbf{Q_1}$ *"Every given specification was easy to read and understand."* According to the obtained answers (see Table 11a), the perceived difficulty was almost equal. This means that most of the participants in

both groups agree that the provided informal descriptions of the software system applications were easily understood.

- **Q_2** *"I had no trouble to specify structural elements of the given informal specifications."* The results in Table 11b show that for the *Traits* group 17 (34.69%) participants rank their expressing of structural properties neutral. Among the other participants, one half tends to strongly agree and the other half to strongly disagree. The *Interfaces* group answers of **Q_2** are more split with the two biggest groups saying they agree and the other one disagrees.
- **Q_3** *"I had no trouble to specify behavioral elements of the given informal specifications."* The answers of this question (see Table 11c) reflect that in both language construct groups the participants had more or less troubles to express behavioral properties, but the results of the behavioral correctness (see Figure 5b) show clearly that the *Traits* group performed way better than the *Interfaces* group.
- **Q_4** *"I had no trouble to specify functionality extensions for the given informal specifications."* Similar to the answers of **Q_3**, Table 11d shows that the participants of the *Traits* group perceived that they had troubles to express functionality extensions (reusable protocol and behavioral properties) but the results for the correctness values of reusability (see Figure 5c) indicate that the *Interfaces* group performed worse than the *Traits* group.
- **Q_5** *"I am familiar with the language concept called Interfaces."* Accordingly to the participants' background information (see Table 4), 100% of them know Java which is more or less reflected in the results to this question (see Table 11e), where we asked the participants if they are familiar with the language construct interfaces.
- **Q_6** *"I am familiar with the language concept called Traits."* In contrast to **Q_5**, the results of this question (see Table 11f) are surprising, because more participants of the *Interfaces* group know the language concept traits compared to the *Traits* experimental group itself. So seemingly the good results for traits have been achieved, even though more knowledge on traits was present in the interfaces group.

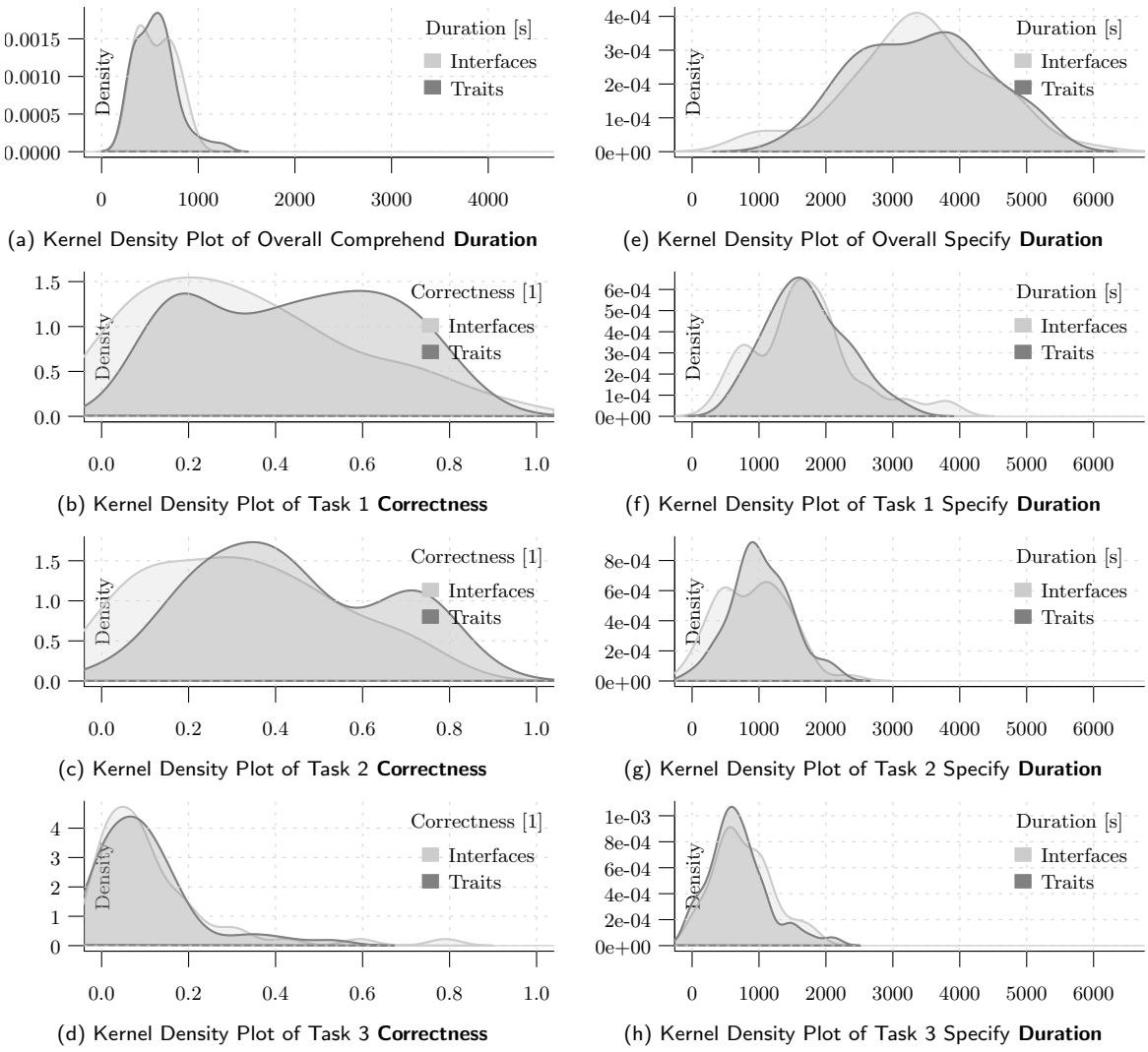
In summary, the post experiment questionnaire shows that the participants believe they understood the constructs to be used reasonably well, and as expected interfaces are better known than traits before the experiment. In this light, our results indicating better results for traits are even more remarkable. It would be interesting to further study how the results would change, if participants would receive training of traits before the experiment.

6.2 Exploration of Moderating Variables

To increase the value of our findings and the resulting conclusions we investigated and explored the following moderating variables – *subject*, *experience*, and *gender*.

Subject. For this moderating variable, we are interested to analyze the participants' task-based performance and if such increases or decreases. In order to obtain such results, we first investigated if there is a difference in the processing time. Due to the experiment design (see Section 3.4), we are able to divide the dependent variable *duration* into two parts – comprehend (reading/understanding) and specify (modeling/writing).

Figure 7a depicts the comprehend *duration* for all tasks whereas Figure 7e depicts the specify *duration* for all tasks. We can observe from those two kernel density plots that the participants spent more time on the actual specifying process than reading and comprehending the informal specification of the given tasks. For



(a) Kernel Density Plot of Overall Comprehend **Duration**

(b) Kernel Density Plot of Task 1 **Correctness**

(c) Kernel Density Plot of Task 2 **Correctness**

(d) Kernel Density Plot of Task 3 **Correctness**

(e) Kernel Density Plot of Overall Specify **Duration**

(f) Kernel Density Plot of Task 1 Specify **Duration**

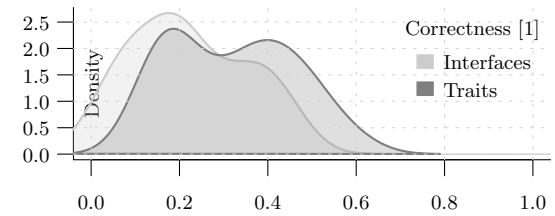
(g) Kernel Density Plot of Task 2 Specify **Duration**

(h) Kernel Density Plot of Task 3 Specify **Duration**

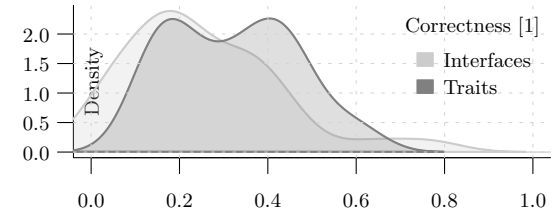
Fig. 7. Descriptive Plots per Group of Overall and per Tasks **Duration** and **Correctness**

both experimental groups the distribution looks very similar. The comprehend and specify duration can be further analyzed for each task. The comprehend duration has a very similar distribution for all three tasks[25]. For the specify duration we can observe a decreasing effect for the processing time which is visualized for Task 1 at Figure 7f, for Task 2 at Figure 7g, and for Task 3 at Figure 7h. This slight decreasing effect of the specify duration can have two origins. Either the participants experience experimental fatigue [61] or a maturation effect [66] took place. In order to analyze those effects we dissected the dependent variable *correctness* for each task – Task 1 at Figure 7b, Task 2 at Figure 7c, and Task 3 at Figure 7d. We can observe that the traits group performs significantly better for Task 1 and Task 2 compared to the interfaces

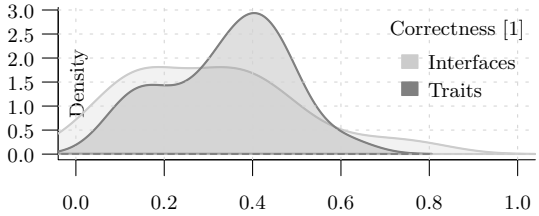
[25]See `appendix.pdf` at [54] for comprehend duration per task plots.



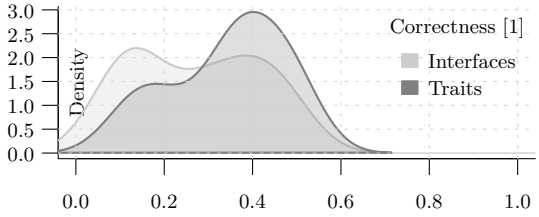
(a) Kernel Density Plot of Less Prog. Exp. **Correctness**



(b) Kernel Density Plot of Less Spec. Exp. **Correctness**



(c) Kernel Density Plot of More Prog. Exp. **Correctness**



(d) Kernel Density Plot of More Spec. Exp. **Correctness**

Fig. 8. Descriptive Plots per Group of **Correctness** by Less/More Experience

group. Despite the shorter specify duration (processing time) in Task 2 the correctness and therefore the participants' performance does not degrade at all. But for Task 3 we can detect a complete drop of the participants' performance for both experimental groups which is the result of experimental fatigue.

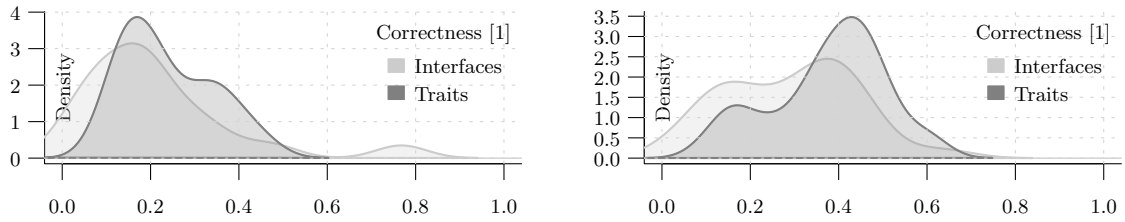
Experience. In order to analyze the moderating variable *experience* we need to determine a classification to separate the obtained experiment samples. Due to the collected background information we can separately analyze a participants' performance in terms of correctness by programming and specifying experience. Therefore, we derive two classifications – *less* experience and *more* experience.

We choose a threshold of 3.25 years in programming experience[26] which results into an exactly equal interfaces to traits sample size ratio for *less* of 28 : 28 and for *more* of 21 : 21. Moreover, we defined that a participant has *less* specifying experience if years <= 2.5. From this it follows that a participant gets classified as *more* experienced if the years > 2.5. This threshold separates the specifying experience[27] with an exactly equal interfaces to traits sample size ratio for *less* of 32 : 32 and for *more* of 17 : 17.

The kernel density plots for programming experience – *less* in Figure 8a and *more* in Figure 8c – as well as the specifying experience – *less* in Figure 8b and *more* in Figure 8d – indicate in all distributions the traits group is performing far better than the interfaces group independently of the classification of their experience. Notable to mention here is that the programming and specifying distributions of the *more* experienced participants achieved a high dense correctness value around 0.4. The latter is an indicator why the traits group is performing better in the overall correctness value despite the number of *more* experienced participants is lower than the number of *less* experienced participants.

Gender. With the moderating variable *gender* we will determine an indicator if one of the experimental treatments does perform in terms of correctness better for a certain gender. According to the obtained

[26] Abbreviated in Figure 8a and Figure 8c as "Prog. Exp.".
[27] Abbreviated in Figure 8b and Figure 8d as "Spec. Exp.".



(a) Kernel Density Plot of Female **Correctness** (b) Kernel Density Plot of Male **Correctness**

Fig. 9. Descriptive Plots per Group of **Correctness** by Gender

participants' background information (see Section 2) the traits to interfaces sample size ratio for females is 20 : 17 and for males is 29 : 32. Since theses numbers are almost equal within a gender we analyzed for each gender the correctness distributions. Figure 9a depicts the kernel density plot for the female correctness whereas Figure 9b depicts the kernel density plot for the male correctness. For both gender the traits group performs slightly better than the interfaces group.

Furthermore, we can observe in Figure 9a and Figure 9b that the participants in this controlled experiment show a clear difference in the performance in terms of correctness depending on the gender. Gren [23] mentions that if there are clear differences in an empirical study based on gender, a proper investigation has to be done to elaborate such effect. By comparing the *gender* results with the data of the *experience* reveals that one possible explanation for the less correct results of the female group can be attributed to lower prior programming experience in the female group compared to the male group.

6.3 Threats to Validity

Threats to Internal Validity. During the experiment, we did not observe any disturbing environmental events or history effects. Due to the total (limited) time of 120 minutes of the experiment, the chances for maturation (carry-over) effects [66] and experimental fatigue [61] were limited. Furthermore, as every participant is only tested once, learning effects can be ruled out. Every participant was able to score the same amount of points and we graded all groups with the same procedures to rule out instrumental bias. Selection bias was limited due to the random assignment of participants to groups. We cannot rule out cross-contamination between the groups as a potential threat to internal validity because the participants are computer science students and share the same social group and interact outside of the research process as well. We have not observed any demoralization or compensatory rivalry. All participants are graded based on their correctness value in the processed survey by gaining points for their enrolled course (but had an opt out option, as explained in Section 3.3).

Threats to External Validity. A possible threat to external validity is that we carried out the experiment with students as participants because this limits the ability to make generalizations. In addition to the types of the participants in this experiment (students as novice software developer or designer), it would be useful to repeat the experiment with broader and more experienced test groups like professionals in different fields ranging from high-level software design to low-level hardware specifications. Furthermore, the selected experiment tasks are limited to basic software system applications. Due to the usage of the syntax keyword `feature`, we mitigated the risk that the participants are biased by identifying language constructs

through known object-oriented abstraction syntax keywords names like `interface` or `trait`. The chosen language construct representations in CASM syntax or their integration into the CASM language might not be representative for potential language constructs and their integration in other ASM languages or other state-based formal languages, and thus our results cannot be generalized to those other languages. We tried to mitigate this threat by only using CASM abstractions that are widely used in other languages, too, and by designing the language constructs as closely as possible to canonical definitions of those abstractions.

Threats to Construct Validity. We focus in this study on the specification effectiveness and efficiency of object-oriented abstractions for an ASM language. The dependent variables *correctness* and *duration* are commonly used to measure the construct **specification effectiveness** and **efficiency**, but other studies use different notations, like Razali et al. [61] which uses *Score (Accuracy)* for **specification effectiveness (correctness)** and *Time Taken* for **specification efficiency (duration)**. Furthermore, other studies analyze both variables under construct names like **comprehensability** (cf. Hoisl et al. [30]) or **understandability** (Czepa et al. [15]). It cannot be ruled out that other constructs would be a better to measure the specification effectiveness and efficiency.

Threats to Content Validity. In this study, we only focus on two object-oriented abstractions, namely interfaces and traits. The specification effectiveness and efficiency is tested for two ASM syntax variations, not commonly existing in today's languages and tools, which use one of the two language constructs (see Section 2.4). Testing more complex scenarios (more complex software system applications and other language constructs) would improve the content validity.

Threats to Conclusion Validity. Due to some missing timestamps for the dependent variable *duration* and unclear written ASM specification solutions for the dependent variable *correctness* we cannot rule out that statistic validity might be affected. Still, those outliers are important measurements because they reflect that for a certain group of the participants the given problem (informal description) to model it through an ASM specification by using a certain language construct are too complex and/or not understood at all. Deleting those would compromise the conclusion validity. To improve the conclusion validity, we selected robust tests with great statistical power which fits the best explored model assumptions of all statistical tests suitable for the collected data set.

6.4 Inferences

Based on the evidence found in this research, a possible use of *Traits* in ASM language designs should provide a good specification effectiveness and efficiency. As *Interfaces* perform significantly worse for the dependent variable **Correctness** than *Traits*, they should be used with more caution. Regarding the dependent variable **Duration**, it seems that for both language constructs the participants need a similar duration to process (read, comprehend, and specify) the tasks and without further studies no generalized claim can be drawn from the gathered results. Taking into account the qualitative measurements, participants using *Traits* without even knowing the language construct specify more efficiently than the *Interfaces* group, which has high familiarity of the language construct (see Section 6). Furthermore, the proposed language syntax of the *Traits*-based ASM specification shows very efficient specification performance for expressing structural and behavioral aspects (see Table 5a and Table 5b) which is not the case for experimental group *Interfaces*.

6.5 Relevance to Practice

So far many formal specification languages lack in their support for other object-oriented language constructs, such as *Interfaces* and *Traits*. As there were no empirical studies on their use in formal specification languages, little was known before this study on how they compare relative to each in the formal methods context.

The findings in this study are first indicators for specification language designers in practice to choose, specify, and implement new language constructs for existing or newly developed programming or specification languages. This could help to create a more understandable *language syntax* which can be used more effectively and efficiently by a *language user* [37]. Many formalisms, including ASMs, are implemented in different programming and/or specification languages. Our empirical results can help specification language designers to choose one of those languages using the available language constructs in the *language syntax* as a decision criterion (among others) and/or by considering the extensibility of the language options with regard to language constructs. The outcome of this study already has made an impact in the state-based formal method community by introducing a *Traits*-based language construct in the CASM language [53].

Due to the fact that the specification effectiveness and efficiency of formal methods has not been empirically investigated to a larger extent so far, these results and future similar empirical studies can contribute to an increased usage of formal methods in practice. Moreover, the explained methods can be used in communities of practice, e.g. by conducting online experiments. The feedback of *language users* is a valuable source for *language engineers* of language extensions and further development.

7 CONCLUSION

This article reports on a controlled experiment with 98 participants on the specification effectiveness and efficiency of the object-oriented abstractions interface and trait, tested for their applicability in the context of state-based formal methods, with ASMs as a representative method. The objective of this study is the investigation on how effective and efficient participants are to specify (express) structural, behavioral, functional, and reusable properties modeled through an ASM-based specification language by using one of the two CASM language syntax extensions, which are not yet part of CASM or any other ASM-based language, namely *Interfaces* and *Traits*.

According to the results of the descriptive and inferential statistics in this study, the experiment group which expresses the given problems through *Traits*-based ASM specifications shows significantly better results in terms of **Correctness** compared to the experiment group which uses *Interfaces*-based ASM specifications. As only one participant has prior knowledge in Rust, only 27 participants have prior knowledge in Scala, but all participants know Java, a higher familiarity with *Interfaces* than with the *Traits* language construct can be assumed for our participants. Nonetheless, in our study results, the specification effectiveness of *Traits* is in terms of the dependent variable **Correctness** significantly better than *Interfaces*, which might be surprising. One explanation of this surprising effect can be drawn by looking at the gathered results of the post experiment questionnaire. Participants from the experimental group *Traits* judge that their understanding of behavioral aspects like extending functionality is similar to the participants of the experimental group *Interfaces*. But the behavioral **correctness** measurement shows that the results are far better in the *Traits* group compared to the *Interfaces* group.

Furthermore, as both object-oriented abstractions perform very similarly in terms of **Duration**, more research is needed to understand the reasons why *Interfaces* perform worse with regard to only one of the two dependent variables. In such a follow-up study an investigation is needed to examine if the specification effectiveness is even better for developers (or professionals) which are highly familiar with *Traits*.

We further analyzed the dependent variable **correctness** according to the evaluation criteria groups – structural, behavioral, reusable, functional, and syntactic, and took into account the qualitative responses of participants. From this, we concluded that the significant difference between the two language constructs is due to the fact that even participants who are not yet familiar with the *traits* language concept specify more effectively with *traits* than participants who use the *interfaces*-based syntax extension and might already know it well.

We believe that this study is the first step towards more understandable and comprehensible ASM language design with regard to object-oriented abstractions for expressing state and behavioral aspects in a maintainable and reusable way. Just like it is the case for CASM, the outcomes of this study can be used by language designers and compiler engineers to define suitable language constructs in other ASM-based languages or state-based formal methods.

It would be interesting to study further our results and complement the statistical analysis with a qualitative analysis of the errors the participants made during the experiment to obtain a more in-depth knowledge how and why there are significant differences in terms of the effectiveness.

ACKNOWLEDGMENTS

We would like to thank all students who participated in this empirical study of the SE2 course in the winter term 2018/19. Furthermore, we want to thank Christoph Czepa for the information and help with statistical procedures and Emmanuel Pescosta for the discussions about object-oriented language abstractions.

REFERENCES

[1] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. 2010. Rodin: an open toolset for modelling and reasoning in Event-B. *International journal on software tools for technology transfer* 12, 6 (2010), 447–466.

[2] Matthias Anlauff. 2000. XASM – An Extensible, Component-based Abstract State Machines Language. In *Abstract State Machines-Theory and Applications*. Springer, 69–90.

[3] Mike Barnett and Wolfram Schulte. 2001. Spying on Components: A Runtime Verification Technique. In *Proceedings of the Workshop on Specification and Verification of Component-Based Systems (SAVCBS'01)*. 7–13.

[4] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the royal statistical society. Series B (Methodological)* (1995), 289–300.

[5] Dines Bjørner. 1979. The vienna development method (VDM). In *Mathematical Studies of Information Processing*. Springer, 326–359.

[6] Fred H Borgen and Mark J Seling. 1978. Uses of discriminant analysis following MANOVA: Multivariate statistics for multivariate purposes. *Journal of Applied Psychology* 63, 6 (1978), 689.

[7] Egon Börger and Alexander Raschke. 2018. *Modeling Companion for Software Practitioners*. Springer.

[8] Egon Börger and Robert Stärk. 2003. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Science & Business Media.

[9] P. S. Canning, W. R. Cook, W. L. Hill, and W. G. Olthoff. 1989. Interfaces for Strongly-typed Object-oriented Programming. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications* (New Orleans, Louisiana, USA) *(OOPSLA '89)*. ACM, New York, NY, USA, 457–467.

[10] Kwang-Ting Cheng and Avinash S Krishnakumar. 1993. Automatic functional test generation using the extended finite state machine model. In *Design Automation, 1993. 30th Conference on*. IEEE, 86–91.

[11] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. 2005. Model variables: Cleanly supporting abstraction in design by contract. *Software: Practice and Experience* 35, 6 (2005), 583–599.

[12] Gastón Christen, Alejandro Dobniewski, and Gabriel Wainer. 2004. Modeling state-based DEVS models in CD++. In *proceedings of MGA, advanced simulation technologies conference*. 105–110.

[13] Edmund M. Clarke and Jeannette M. Wing. 1996. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv.* 28, 4 (Dec. 1996), 626–643. https://doi.org/10.1145/242223.242257

[14] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993), 494.

[15] Christoph Czepa, Huy Tran, Uwe Zdun, Thanh Tran Thi Kim, Erhard Weiss, and Christoph Ruhsam. 2017. On the Understandability of Semantic Constraints for Behavioral Software Architecture Compliance: A Controlled Experiment. In *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, 155–164.

[16] Christoph Czepa and Uwe Zdun. 2018. On the Understandability of Temporal Properties Formalized in Linear Temporal Logic, Property Specification Patterns and Event Processing Language. *IEEE Transactions on Software Engineering* (2018).

[17] Luca De Alfaro and Thomas A Henzinger. 2001. Interface theories for component-based design. In *International Workshop on Embedded Software*. Springer, 148–165.

[18] Olive Jean Dunn. 1958. Estimation of the means of dependent variables. *The Annals of Mathematical Statistics* (1958), 1095–1111.

[19] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. 2007. CoreASM: An Extensible ASM Execution Engine. *Fundamenta Informaticae* 77, 1-2 (2007), 71–104.

[20] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. 2008. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *Journal of Universal Computer Science* 14, 12 (2008), 1949–1983.

[21] David Garlan. 2003. Formal modeling and analysis of software architecture: Components, connectors, and events. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 1–24.

[22] Uwe Glässer and Margus Veanes. 2002. Universal Plug and Play Machine Models. In *Design and Analysis of Distributed Embedded Systems*. Springer, 21–30.

[23] Lucas Gren. 2018. On Gender, Ethnicity, and Culture in Empirical Software Engineering Research. In *2018 IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 77–78.

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

[25] Yuri Gurevich. 2000. Sequential Abstract-State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic (TOCL)* 1, 1 (2000), 77–111.

[26] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. 2004. Semantic Essence of AsmL. In *Formal Methods for Components and Objects*. Springer, 240–259.

[27] Yuri Gurevich and Nikolai Tillmann. 2001. Partial Updates: Exploration. *Journal of Universal Computer Science* 7, 11 (2001), 917–951.

[28] David Harel and Amnon Naamad. 1996. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 4 (1996), 293–333.

[29] Andreas Höfer and Walter F Tichy. 2007. Status of empirical research in software engineering. In *Empirical Software Engineering Issues. Critical Assessment and Future Directions*. Springer, 10–19.

[30] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. 2014. Comparing three notations for defining scenario-based model tests: A controlled experiment. In *Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the*. IEEE, 180–189.

[31] James K Huggins and David Van Campenhout. 1998. Specification and verification of pipelining in the ARM2 RISC microprocessor. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 3, 4 (1998), 563–580.

[32] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 2 (2002), 256–290.

[33] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. 2008. Reporting Experiments in Software Engineering. In *Guide to advanced empirical software engineering*. Springer, 201–228.

[34] Natalia Juristo and Ana M Moreno. 2013. *Basics of software engineering experimentation*. Springer Science & Business Media.

[35] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. 2017. Robust Statistical Methods for Empirical Software Engineering. *Empirical Software Engineering* 22, 2 (2017), 579–630.

[36] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering (TSE)* 28, 8 (2002), 721–734.

[37] Anneke G. Kleppe. 2009. Software Language Engineering: Creating Domain-Specific Languages using Metamodels. *Addisson-Wesley* 33, 1 (2009).

[38] William H Kruskal and W Allen Wallis. 1952. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association* 47, 260 (1952), 583–621.

[39] Leslie Lamport. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 872–923.

[40] Kevin Lano. 1991. Z++, an Object-Orientated Extension to Z. In *Z User Workshop, Oxford 1990*. Springer, 151–172.

[41] Roland Lezuo. 2014. *Scalable Translation Validation; Tools, Techniques and Framework*. Ph.D. Dissertation. Wien, Techn. Univ., Diss.

[42] Roland Lezuo, Gergö Barany, and Andreas Krall. 2013. CASM: Implementing an Abstract State Machine based Programming Language. In *Software Engineering (Workshops)*. 75–90.

[43] Roland Lezuo, Philipp Paulweber, and Andreas Krall. 2014. CASM - Optimized Compilation of Abstract State Machines. In *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*. ACM, 13–22.

[44] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).

[45] Barbara Liskov and Stephen Zilles. 1974. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages* (Santa Monica, California, USA). ACM, New York, NY, USA, 50–59. https://doi.org/10.1145/800233.807045

[46] Robert C Martin. 1997. Java and C++ A critical comparison. *Technical Note, Object Mentor* (1997).

[47] Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

[48] Emerson R. Murphy-Hill, Philip J. Quitslund, and Andrew P. Black. 2005. Removing Duplication from Java.Io: A Case Study Using Traits. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. ACM, New York, NY, USA, 282–291. https://doi.org/10.1145/1094855.1094963

[49] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala*. Artima Inc.

[50] Flavio Oquendo. 2004. π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes* 29, 3 (2004), 1–14.

[51] June Jamrich Parsons. 2012. *Practical Open Source Office: LibreOffice(TM) and Apache OpenOffice* (2nd ed.). Course Technology Press, Boston, MA, United States.

[52] Philipp Paulweber, Emmanuel Pescosta, and Uwe Zdun. 2018. 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)*. Springer, 39–54.

[53] Philipp Paulweber, Emmanuel Pescosta, and Uwe Zdun. 2020. Structuring the State and Behavior of ASMs: Introducing a Trait-Based Construct for Abstract State Machine Languages. In *International Conference on Rigorous State-Based Methods (Lecture Notes in Computer Science 12071)*. Springer, 237–243.

[54] Philipp Paulweber, Georg Simhandl, and Uwe Zdun. 2021. Specifying with Interface and Trait Abstractions in Abstract State Machines: A Controlled Experiment. https://doi.org/10.5281/zenodo.4517172. Data-Set and Artifacts: Documents, Forms, and R Scripts for Reproducibility of the Empirical Study.

[55] Philipp Paulweber, Georg Simhandl, and Uwe Zdun. (under major revision). On the Understandability of Language Constructs to Structure the State and Behavior in Abstract State Machine Specifications: A Controlled Experiment. submitted paper to JSS.

[56] Philipp Paulweber and Uwe Zdun. 2016. A Model-Based Transformation Approach to Reuse and Retarget CASM Specifications. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016 (Lecture Notes in Computer Science 9675)*. Springer, 250–255.

[57] Ben Potter, David Till, and Jane Sinclair. 1996. *An introduction to formal specification and Z*. Prentice Hall PTR.

[58] Anthony Potts and David H Friedel. 2018. *Java programming language handbook*. Coriolis Group Books.

[59] R Development Core Team. 2008. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

[60] Alexander Raschke, Dominique Méry, and Frank Houdek. 2020. Rigorous State-Based Methods. In *Proceedings of 7th International Conference, ABZ 2020, Ulm, Germany, May 27–29, 2020*. Springer, 8.

[61] Rozilawati Razali, Colin F Snook, Michael R Poppleton, Paul W Garratt, and Robert Walters. 2007. Experimental Comparison of the Comprehensibility of a UML-based Formal Specification versus a Textual One. In *11th International*

Conference on Evaluation and Assessment in Software Engineering (EASE) 11. 1–11.

[62] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments?. In *37th IEEE/ACM International Conference on Software Engineering (ICSE)*, Vol. 1. IEEE, 666–676.

[63] Hisashi Sasaki. 1999. A Formal Semantics for Verilog-VHDL Simulation Interoperability by Abstract State Machine. In *Proceedings of the Conference on Design, Automation and Test in Europe* (Munich, Germany) *(DATE '99)*. ACM, New York, NY, USA, Article 73.

[64] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. 2003. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*. Springer, 248–274.

[65] Joachim Schmid. 2001. Introduction to AsmGofer. *http://www.tydo.de/AsmGofer* (2001).

[66] SJ Senn. 1992. Is the 'simple carry-over'model useful? *Statistics in Medicine* 11, 6 (1992), 715–726.

[67] Y. Shafranovich. 2005. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180.

[68] Samuel Sanford Shapiro and Martin B Wilk. 1965. An analysis of variance test for normality. *Biometrika* 52, 3/4 (1965), 591–611.

[69] Graeme Smith. 2012. *The Object-Z Specification Language*. Vol. 1. Springer Science & Business Media.

[70] Colin Snook and Rachel Harrison. 2001. Practitioners' views on the use of formal methods: an industrial survey by structured interview. *Information and Software Technology* 43, 4 (2001), 275–283.

[71] Ann E Kelley Sobel and Michael R Clarkson. 2002. Formal methods application: An empirical tale of software development. *IEEE Transactions on Software Engineering* 28, 3 (2002), 308–320.

[72] Robert F Stärk, Joachim Schmid, and Egon Börger. 2001. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer Berlin Heidelberg.

[73] Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. 2008. Using students as subjects-an empirical evaluation. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 288–290.

[74] Rini Van Solingen, Vic Basili, Gianluigi Caldiera, and H Dieter Rombach. 2002. Goal Question Metric (GQM) Approach. *Encyclopedia of software engineering* (2002).

[75] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer Science & Business Media.