On the Understandability of Language Constructs to Structure the State and Behavior in Abstract State Machine Specifications: A Controlled Experiment

Philipp Paulweber*, Georg Simhandl, Uwe Zdun

University of Vienna, Faculty of Computer Science, Research Group Software Architecture, Währingerstraße 29, 1090 Vienna, Austria

Abstract

Abstract State Machine (ASM) theory is a well-known state-based formal method to analyze and specify software and hardware systems. As in other state-based formal methods, the proposed modeling languages for ASMs still lack easy-to-comprehend abstractions to structure state and behavior aspects of specifications. Modern object-oriented languages offer a variety of advanced language constructs, and most of them either offer interfaces, mixins, or traits in addition to classes and inheritance. Our goal is to investigate these language constructs in the context of state-based formal methods using ASMs as a representative of this kind of formal methods. We report on a controlled experiment with 105 participants to study the understandability of the three language constructs in the context of ASMs. Our hypotheses are influenced by the debate of object-oriented communities. We hypothesized that the understandability (measured by correctness and duration variables) shows significantly better understanding for interfaces and traits compared to mixins, as well as at least a similar or better understanding for traits compared to interfaces. The results indicate that understandability of interfaces and traits show a similar good understanding, whereas mixins shows a poorer understanding. We found a significant difference for the correctness of understanding when comparing interfaces and mixins.

Keywords: Abstract State Machines, Empirical Software Engineering, Understandability, Language Constructs, Controlled Experiment

1. Introduction

The Abstract State Machine (ASM) theory is a well-known state-based formal method consisting of transition rules and algebraic functions and has been described by Gurevich in the beginning of the 1990s. Several scientists of different research fields used and applied the ASM theory and its method. [1, 2].
This usage ranges from software, hardware and system engineering perspectives to specify, analyze, verify, and validate systems in a formal way \[3\]. The diversity of ASM-based applications ranges from formal specification of semantics of programming languages, such as those for Java by Stäck et al. \[4\] or Very High Speed Integrated Circuit Hardware Description Language (VHDL) by Sasaki \[5\], compiler back-end verification by Lezuo \[6\], software run-time verification by Barnett and Schulte \[7\], software and hardware architecture modeling e.g. of Universal Plug and Play (UPnP) by Glässer and Veanes \[8\], to even Reduced Instruction Set Computing (RISC) designs by Huggins and Campenhout \[9\].

By definition, an ASM formally describes the evolution of function states through dedicated transaction rules in a step-by-step manner \[2\] and are used to specify sequential, parallel, concurrent, reflective, and even quantum algorithms \[2\]. In order to describe, analyze, and even execute ASMs, several languages and tools were developed over time to model ASM specifications based on the ASM theory description by Gurevich \[1\]. Additionally, several theory improvements were provided to increase the expressiveness of ASM languages which were summarized by Börger and Stäck \[2\] and Börger and Raschke \[10\].

The landscape of developed ASM languages and the corresponding tools is rather limited nowadays. Best known are the ASM implementations AsmetaL \[11\] and CoreASM \[12,13\]. AsmetaL provides a feature-rich tool set to model, analyze, interpret, and generate code of described ASM specifications\[12\]. The core of AsmetaL is implemented and based on the Eclipse Modeling Framework (EMF) and provides therefore a Java-based interpreter. CoreASM\[3\] is another Java-based interpreter implementation for ASM specifications. Its main focus is on the language extensibility which is supported through the adoption of the parser implementation \[12\]. The base implementation CoreASM is written in Java as well as all language extensions have to be written in Java. Besides this two interpreter-oriented implementations there exists AsmL \[13\] and Corinthian Abstract State Machine (CASM) \[14\]. AsmL is based on the .NET framework and allowed the compilation (code generation) of ASM specifications. Gurevich itself was part of this project but it discontinued. CASM\[4\] was introduced by Lezuo and provides compilation as well as interpreting of modeled ASM specifications. Due to the compilation focus of CASM it uses a statically typed inferred language design and Lezuo et al. \[14\] established compilation techniques to outperform CoreASM and AsmL in terms of ASM execution performance. There are several other ASM language tool implementations like AsmGofer \[15\] or eXtensible ASM (XASM) \[16\], but those projects are discontinued.

ASMs are part of the state-based formal methods which provide their own languages and tools. The most prominent candidates are Alloy \[17\], Event-B \[18\], Temporal Logic of Actions (TLA) \[19\], Vienna Development Method (VDM) \[20\], and Z \[21\].

---

1. The ASM theory was formerly called Evolving Algebra.
2. See https://asmeta.github.io for the AsmetaL project site.
3. See https://github.com/CoreASM for the CoreASM project site.
1.1. Problem Statement

Today, a common threat in the various ASM languages and tools, as well as in most other state-based formal methods, is that the proposed modeling languages lack easy to comprehend abstractions to describe reusable and maintainable specifications. While very few have embraced basic object-oriented abstractions such as classes and inheritance, more advanced language constructs are usually missing. Mernik et al. point out that the lack of such object-oriented abstractions in formal methods is one of the main reasons why formal methods and their languages are not widely used and are more or less unpopular compared to feature-rich programming languages. Börger suggests in one of his latest articles that we need better abstractions (language constructs) in existing ASM modeling languages without focusing on class and inheritance concepts.

In contrast modern object-oriented languages offer a variety of advanced language constructs, and most offer either interfaces, mixins, or traits in addition to classes and inheritance. All of those three language constructs have similar and some different properties and characteristics, which are depicted in Figure 1 and described as follows:

**Interfaces** define (typed) operations (signatures) to which an implementer of a certain interface (type) has to conform. Therefore, an interface defines a so-called contract. No behavioral or state information can be defined through interfaces.

**Mixins** can define reusable behavioral and state information that can be used to combine (mix) and form new types. Mixins enrich interfaces with behavioral and state information.

**Traits** are similar to interfaces with the difference that they can define stateless behavior which depends on the trait itself. Therefore, compared to mixins, a definition of a state in a trait is not allowed. The properties and capabilities of traits are situated between the other language constructs interfaces and mixins.

There is a heated debate in the object-oriented community, which of those abstractions is best suited to promote reusable and maintainable specifications, and many implementations combine different language constructs. A notable example would be the programming language Scala, which offers a trait
syntax that is similar to the Java 8 interface syntax and offers mixins language constructs through the class-based implementation and extension syntax. Another example of mixed language constructs, namely interfaces and traits, can be found in the programming language Rust, where the language user has to express every interface definition through traits and the structures (as well as types) have to conform to specified traits and implement all required functionalities.

Empirical research on language constructs in ASM languages and similar state-based formal methods has the potential to influence language designers and compiler engineers when making decisions on choosing language constructs in specification language designs and implementations.

1.2. Research Objectives, Hypotheses, and Results

In this empirical study we investigate how well and fast a participant understands textual language construct representations for state-based formal methods. State-based formal methods and their modeling languages are usually based on base concepts that are significantly different from classes. 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 as a representative for ASM-based languages and tools. As our study focuses on the general notion of adding advanced language constructs to CASM, we believe that most of our results can be generalized to other ASM languages and tools. The latter could be confirmed with a follow-up study.

In this study the term understandability corresponds to how well and fast a participant understands a given language construct in example ASM specifications. We define the experiment goal using the Goal Question Metric (GQM) template as follows: Analyze the Interfaces, Mixins, and Traits language constructs for the purpose of their evaluation with respect to their understandability from the viewpoint of the novice and moderately advanced software architect, designer, or developer in the context (environment) of the Advanced Software Engineering (ASE) and Distributed Systems Engineering (DSE) courses at the Faculty of Computer Science of the University of Vienna.

Our hypotheses are influenced by the debate in the object-oriented community, which recently discuss traits often more favorably than mixins. In particular, mixins contain state information whereas traits do not, mixins use implicit conflict resolution whereas traits use explicit resolution and mixins are linearized (order of used language construct matters) whereas traits are flattened (order of used language construct does not matter). Also, the community often discusses traits more favorably than interfaces or point out that “Traits are In-

---

4See https://cs.univie.ac.at for faculty website.

terfaces” with code-level reuse functionality. On the other hand, 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 and abstract class in C++. As a consequence, we hypothesized that understandability measured by correctness and duration variables shows a significantly better understanding for traits compared to mixins and for interfaces compared to mixins. Further, we derived from the debate another hypothesis that traits offer at least a similar or even better understanding compared to interfaces.

The obtained results in this study indicate that the language constructs interfaces and traits show a similar good understanding. The language construct mixins shows poorer understanding compared to interfaces and traits, which indicates that from a language user perspective the strict separation of behavioral and structural elements is better understandable than the intermixed representation form.

1.3. Structure of this Article

In Section 2, we describe ASMs, the used language and constructs used in this study, and present related studies. Section 3 elaborates the planning of the language construct study. In Section 4, we describe the execution of the experiment, while the results are presented in Section 5 and discussed in Section 6. Last but not least, we conclude the article in Section 7.

2. Background

This section discusses some properties regarding ASMs and language constructs that are of interest in this study. Readers already familiar with ASMs and the discussed type abstractions and their corresponding representations may consider to skip the whole or some parts of this section.

2.1. Abstract State Machines

ASMs are used to express calculations in an abstract manner for all kind of different application fields. According to Gurevich and Tillmann [37], the ASM thesis states that if there is a computer system $A$, it can be simulated in a step-by-step manner by a behavioral 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) stepwise refinement of the reference model by instantiating more concrete models which uphold the properties of the reference model [2].

---

7See, e.g. [https://blog.rust-lang.org/2015/05/11/traits.html](https://blog.rust-lang.org/2015/05/11/traits.html)
ASMs has two field 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 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, every state information in an ASM specification is expressed through a function definition (see Section 2.2).

**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. ASM rules allow interleaved parallel and sequential execution semantics, a correct ASM specification does not allow the update (insertion to the update-set) of the same function location twice or more, which is referred in the literature as an inconsistent update. A language user can model transactions through named rules (see Section 2.2).

**Agents** are the actors of an ASM specification. There can be one (single) agent or multiple agents. Every agent activates his 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 makes the usage replacing 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 the usage of upgrading the functionalities or changing the environmental settings. A vertical refinement adds more and more details about the application or system (e.g. add another requirement, more states etc.). A more detailed description and elaboration of the ASM modeling and refinement concepts is given by Börger and Raschke.

### 2.2. ASM Language Representative

In this study, we use the basic syntax elements from the CASM language. 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 generalized to these other ASM languages and also to other state-based formalisms.

---

8See [https://casm-lang.org/syntax](https://casm-lang.org/syntax) for CASM language description.
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, deriveds are side-effect free functions and can be in some cases even pure functions. Listing 2 illustrates the concrete syntax and some examples which use state information from Listing 1.

**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 through basic ASM rules which are: (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 non-deterministic 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 [10]. Listing 3 illustrates the concrete syntax and an example which depends on some definitions from Listing 1 and Listing 2.

2.3. Experiment Language Construct Representations

Besides a class concept used in AsmL [13], no other advanced language construct has been introduced in the ASM language and tool landscape. To enable moving the state-of-the-art in advanced language constructs for such formal languages forward, this study tests three representations of language constructs, namely interfaces, mixins, and traits, to search for a suitable language construct, structuring and extension of functionality for such languages in general and specifically forCASMIN. 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.

Please note that we use these very general terms on purpose as they can be mapped to all three language constructs under investigation. As a consequence, we can avoid that participants in the experiment are biased by knowing keywords identifying the language construct through interface, mixin, or trait which especially applies for the keyword feature. All three language construct syntax 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 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. This syntax is primarily influenced by the Java programming language interface syntax.

---

9See form_ifaces.pdf at [40].
**Language Construct Mixins (Experiment Group C)**

The feature syntax in the language construct Mixins is equal to Interfaces except that it supports an optional default implementation through an implement definition. Besides the default behavior such a definition can define an internal state through function definitions. Therefore, mixins can define required type behavior and state. To indicate that a structure shall provide the behavior of a feature, the implement keyword is used to extend the current structure implementation by the default implementation and function state. Every default implementation can be overwritten by an explicit concrete implementation of a certain operation. Listing 5 depicts an example specification with the Mixins language construct. This syntax is primarily influenced by the Scala programming language trait syntax which enables mixins capabilities.

**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) provides the behavior of the structure; (2) provides 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 6 depicts an example specification with the Traits language construct. This feature and implement syntax is influenced by the Rust programming language trait syntax.

2.4. Related Studies

So far, interfaces, mixins 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. Apel and Batory present a case study comparing aspect and feature abstractions using a mixin layer approach to unify the two. Batory et al. present another case study on achieving extensibility through product-lines and domain-specific languages using a mixin-based approach. However, so far no study comparing the three advanced language constructs covered in our study exists and also no controlled experiments.

Interface abstractions have been extensively studied in the context of formal methods and architecture description languages that offer formal representations. Traits and mixins, in contrast have not yet been studied in the context of formal methods. We are not aware of any formal method

---

10See form_mixins.pdf at [40].
11See form_traits.pdf at [40].
12See https://doc.rust-lang.org/rust-by-example/trait.html for the discussion.
Listing 4: *Interfaces*-Based Example Specification

```
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
  // encapsulated feature implementation
  derived toString : String = this.getName() + ( this.getAge() as String )
}
```

Listing 5: *Mixins*-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
  // overwite of feature implementation
  derived toString : String = this.getName() + ( this.getAge() as String )
}
```

Listing 6: *Traits*-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
  derive toString : String = this.getName() + ( this.getAge() as String )
}
```
that unifies or integrates any two or all three advanced 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 [50]. Czepa and Zdun [51] and Czepa et al. [52] have studied the understandability of formal methods for temporal property specification using similar research methods as used in this study.

Ferrarotti et al. [53] report on a recent study where ASM-based high-level software specifications are extracted from Java programs by using an semi-automated approach. This study is of interest, because it maps the Java object-oriented programming language concepts to the ASM sub-machine [53] concept in order to represent the abstract type (interfaces) and sub-classing mechanisms.

Related to this study, we conducted another controlled experiment [54] with 98 participants where we analyzed the specification efficiency by using only the language constructs interfaces and traits. Since this study only investigates how well participants can understand (read, comprehend) ASM specifications by answering questions about certain properties, the other study [54] investigates how efficient and effective participants can write (specify) ASM specifications using a certain language construct and receiving an informal system description as stimuli. The results indicate that the language construct trait is more efficient than interfaces. Apart from that, we are not aware of any other empirical study that systematically investigated advanced language constructs in the context of formal methods.

3. Experiment Planning

This study is structured following the guidelines by Jedlitschka et al. [55] on how empirical research shall be conducted and reported in software engineering. Moreover, the guidelines by Kitchenham et al. [56], Wohlin et al. [57], and Juristo and Moreno [58] 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. [59].

3.1. Goals

The goal of this experiment is to measure the construct understandability on how well and fast a participant understands a given textual representation of three different language constructs, namely Interfaces, Mixins, and Traits. The quality focus of the construct understandability is the correctness and duration of the participant’s answers.

3.2. Context and Design

This study reports on a controlled experiment with 105 participants in total to study the understandability of the language constructs interfaces,
mixins, 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 textual representation of applications described in an ASM language and how well and correct the participant answers behavioral and structural questions about the given applications.

The study was carried out with 70 computer science students who had enrolled in the course ASE\(^{13}\) which is a mandatory part of the Master of Science (MSc) curricula at the University of Vienna, and with 35 computer science students who had enrolled in the course DSE\(^{14}\) which is an optional part of the Bachelor of Science (BSc) and MSc curricula at the University of Vienna, at the same time respectively in the summer term 2018. All participants had a limited time of 105 minutes to process the survey.

3.3. Participants

All participants of the experiment are BSc and MSc students of the Faculty of Computer Science at the University of Vienna, Austria enrolled in at least one of the following courses:

- DSE: BSc and MSc students are enrolled in the course and used as proxies for novice to moderately advanced software architects, designers, or developers. This course, which is intended for students in the fourth semester of the BSc curricula or first semester of the MSc curricula, is concerned with teaching principles of distributed systems, programming and engineering methods for distributed software, and solving accompanying problems like latency, concurrency, unpredictability, and scalability.

- ASEL: MSc students are enrolled in the course and used as proxies for moderately advanced software architects, designers, or developers. This course, which is intended for students in the second semester of the MSc curricula, is concerned with teaching principles of modern software engineering methods, including distributed software architectures, design methods, and advanced software engineering tools and techniques for Domain Specific Language (DSL) \(^{60}\) and Model-Driven Development (MDD) \(^{61}\) approaches.

For both courses, the participants (students) received training in programming, software engineering, (data) modeling, basic formal methods, algorithms, and mathematics. At the beginning of the courses, 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)

\(^{13}\)See [https://ufind.univie.ac.at/en/course.html?lv=053020&semester=2018S](https://ufind.univie.ac.at/en/course.html?lv=053020&semester=2018S) for ASE.

\(^{14}\)See [https://ufind.univie.ac.at/en/course.html?lv=052500&semester=2018S](https://ufind.univie.ac.at/en/course.html?lv=052500&semester=2018S) for DSE.
a participant answered the survey determined the bonus points achieved (for \textit{correctness} definition, see Section 3.5).

In total, there were 105 participants, which were randomly allocated to the treatments (i.e. the three language construct representations in an \textit{ASM} specification language, see Section 2). Due to random assignment of the participants to groups – \textit{Interfaces} (Group A), \textit{Mixins} (Group C), and \textit{Traits} (Group B) – the final distribution resulted in 36 : 34 : 35.

Someone may argue that students as experiment participants are not good proxies for novice and moderately advanced software engineers. The participants in our experiment are students of two advanced courses \textit{DSE} and \textit{ASE} 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. \cite{56} using students “\textit{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 2c, which summarizes the participants’ industrial work experiences).

Other studies by Svahnberg et al. \cite{62} or Salman et al. \cite{63} 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 design patterns for distributed system applications. The selection includes the \textit{Message Queue}, \textit{Publish-Subscribe}, and \textit{Remote Procedure Call} patterns as example applications inspired by examples provided by Börger and Raschke \cite{10}.

The selected software design patterns are related to the subjects taught in both courses – \textit{DSE} and \textit{ASE}. This study consists of two major experiment material artifacts:

\begin{enumerate}
\item \textbf{Information Sheet} An experiment information document\footnote{See \texttt{info.pdf} at \cite{10}.} explaining the \textit{ASM} language syntax and semantics \textbf{without the experiments’ language construct} syntax and semantics extensions.
\end{enumerate}
(2) Survey Form Three experiment survey forms per experimental group and language construct contain the actual survey along with the explicit experiments’ language construct syntax and semantics extension and description per experimental group.

All three experiment survey forms are structured the same way consisting of four parts: (1) a participant information questionnaire; (2) the experiments’ group language construct syntax and semantics extension description; (3) three experiment tasks (equal to all experiment groups); (4) an overall experiment questionnaire.

Each experiment task consists of a given ASM specification, which is provided in the different experiment groups in the respective language construct (Interfaces, Mixins, or Traits) textual representation. Every task is divided into sub-tasks to test the participants’ understandability of the given ASM specification. The students (participants) were instructed to read the given ASM specification before they start to process the following four sub-tasks:

(1) Behavioral Four yes-and-no questions were used to determine understanding of behavioral properties. An example question in task 2: “A Service can only handle structure values, which implement the Subscriber feature”.

(2) Structural Four filling-out-blanks sentences were used to determine understanding of structural properties. An example sentence in task 2: “The feature _____ is implemented (included) two times for a structure.”

(3) Operational Multiple-choice answers of console outputs were used to determine understanding of operational and executable properties of the given ASM specification.

(4) Self Assessment A task-based questionnaire was used to obtain an objective perspective of the participants’ self assessment of how correct their answers are with a certain level of confidence.

Important is that all the sub-tasks (questions) are identical except for the textual representation of the given ASM specification in the corresponding experiment groups’ language construct.

3.5. Variables and Hypotheses

This controlled experiment measures the following two dependent variables:

(1) Correctness as achieved in answering the questions, which include trying to mark the correct answer and filling in the blanks in the tasks;

(2) Duration as the time it took to answer the questions of all tasks in an experiment survey form (see Section 3.4) excluding breaks.

---

16See form_ifaces.pdf, form_mixins.pdf, and form_traits.pdf at [here].
These two dependent variables are commonly used to measure the construct **understandability** (cf. Hoisl et al. [64], Czepa and Zdun [51]). The independent variables (factors) have three treatments, namely the three different representations of language constructs **Interfaces**, **Mixins**, and **Traits**.

We hypothesized that **Traits** are easier to understand than **Mixins** due to the explicit and separated functionality extension definition blocks offered by traits. And **Interfaces** are easier to understand than **Mixins** due to their simplicity without the additional overhead of possible default implementations and optional local state bound to a certain type.

Furthermore, we hypothesized that **Traits** are easier to understand than or as understandable as **Interfaces** due to their almost equal Application Programming Interface (API) declaration styles. Consequently, we formulate the following null hypotheses, where understandability is measured by correctness and duration variables, for this controlled experiment:

**H_{0,1}** There is no difference in terms of understandability between **Interfaces** and **Mixins**.

**H_{0,2}** There is no difference in terms of understandability between **Traits** and **Mixins**.

**H_{0,3}** There is no difference in terms of understandability between **Interfaces** and **Traits**.

Based on the formulated null hypotheses, we can derive and formulate the following alternative hypotheses for this controlled experiment:

**H_{A,1}** The understandability shows a significantly better understanding of **Interfaces** compared to **Mixins**.

**H_{A,2}** The understandability shows a significantly better understanding of **Traits** compared to **Mixins**.

**H_{A,3}** The understandability shows a significantly better or similar understanding of **Interfaces** compared to **Traits**.

4. Experiment Execution

This experiment was executed in two steps, namely 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[^17]. This document provided general information of the upcoming experiment and an introduction to the [ASM] language syntax and semantics used.

[^17]: See [https://moodle.org](https://moodle.org) for e-learning platform information.
without explaining one of the three 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).

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 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’ answer was recorded in a LibreOffice OpenDocument Spreadsheet (ODS) file [65]. 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.

4.3. Deviations

The experiment execution and the data collection were performed as described in Section 4.1 and Section 4.2. We did not observe any unforeseen difficulties and did not deviate from the experiment plan.

5. Analysis

All statistical analysis was performed with the software tool R [19]. The analysis processes 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 [21].

---

18See https://www.libreoffice.org for version 6.1.4.2.
19See https://www.r-project.org for version 3.5.2.
20See analyze.r at [40].
21See install.r at [40].
5.1. Data-Set Preparation

The raw data collected during the experiment execution phase (see Section 4) was prepared in the following manner: (1) the obtained LibreOffice ODS file was exported to a Comma-Separated Values (CSV) file; (2) the CSV file was imported for further processing; (3) type castings of several data rows were performed; (4) overall correctness $C$ of all task correctness values $C_1$, $C_2$, and $C_3$ is obtained by the following formula $C = \sum_{n=1}^{3} C_n \times n$, which means that we weighted the first task correctness $C_1$ with $\frac{1}{6}$, the second task correctness $C_2$ with $\frac{2}{6}$, and the third task correctness $C_3$ with $\frac{3}{6}$ of the overall task correctness $C$ to represent a complexity gain in understanding the given ASM specifications. Every task correctness $C_n$ where $n = 1, 2, 3$ is determined by accumulating the percentage of the correct answers of the sub-tasks 1), 2), and 3) which were explained in Section 3.4 (5) and stored as an R Data-Set (RDS) file for further processing and analysis.

5.2. Descriptive Statistics

The participants’ experience and characteristics (background information) are captured in the experiment by: age (see Figure 2a), gender, course, and level of education, programming experience (see Figure 2b), modeling experience, software (SW) and hardware (HW) industry experience (see Figure 2c), and programming and specification languages used. Overall, the random distribution of the participants to the experiment groups is almost balanced.

The participants’ programming experience (see Figure 2b) refers to the amount of years using one or multiple programming languages either in an industrial work context or an educational work environment or both.

Table 1 contains the number of observations, central tendency measures, and dispersion measures per language construct for the dependent variable Correctness and this acquired data is visualized as a kernel density plot in Figure 3a, and a box plot in Figure 3b. In the box plot we can observe that for the Interfaces group the median and its quantiles are above those of the other groups. There is one outlier in the Mixins group. Note that the Traits group has almost a similar median to the Interfaces group and that this distribution is strongly right skewed. According to the kernel density plot, the data does not appear to be normally distributed, and all three distributions look different, which implies unequal variances. The Interfaces has its peak at 0.55 and Mixins has its peak at 0.45. In contrast to the two other groups, the Traits groups has two peaks, one at about 0.215 and the other one at about 0.525.

---

22The data-set is published in the long term open data archive Zenodo together with all documents and R scripts.
23See prepare.r at [40].
24For detailed formula, see prepare.r Line 235-340 at [40].
25See appendix.pdf at [40] for supplementary background information.
26Unit is correctness rate between 0.0 and 1.0 (denoted [1]).
Table 2 contains the number of observations, central tendency measures, and dispersion measures per language construct for the dependent variable **Duration** and this acquired data is visualized as a kernel density plot in Figure 4a and a box plot in Figure 4b. In the box plot we can observe that for the **Traits** group has the lowest median compared to the other groups, but the quantiles of the **Traits** group are similar to the **Interfaces** group in contrast to the **Mixins** group. According to the kernel density plot, the data does not appear to be normally distributed, and all three distributions look different, which implies unequal variances. The **Traits** group has its peak at 2500 seconds and the **Mixins** group has its peak at 2750 seconds. In contrast to the two other groups, the **Interfaces** group has two peaks, one very flat one at about 2250 seconds and another much bigger one at about 3125 seconds.

### 5.3. Hypothesis Testing

Due to the presence of three experiment groups and two dependent variables, the Multivariate Analysis of Variance (MANOVA) would be a suitable statistical procedure, but necessary assumptions must be met to apply this method.

---

27Unit is duration in seconds (denoted [s]).
Table 1: Descriptive Statistics per Group of Dependent Variable Correctness

<table>
<thead>
<tr>
<th></th>
<th>Interfaces</th>
<th>Mixins</th>
<th>Traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observations</td>
<td>36</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>Mean</td>
<td>0.5294</td>
<td>0.4574</td>
<td>0.4598</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.1110</td>
<td>0.1147</td>
<td>0.1533</td>
</tr>
<tr>
<td>Median</td>
<td>0.5448</td>
<td>0.4707</td>
<td>0.5231</td>
</tr>
<tr>
<td>Median abs. deviation</td>
<td>0.0869</td>
<td>0.0950</td>
<td>0.1030</td>
</tr>
<tr>
<td>Minimum</td>
<td>0.2639</td>
<td>0.1528</td>
<td>0.1204</td>
</tr>
<tr>
<td>Maximum</td>
<td>0.7083</td>
<td>0.6759</td>
<td>0.6528</td>
</tr>
<tr>
<td>Skew</td>
<td>-0.6132</td>
<td>-0.4984</td>
<td>-0.7713</td>
</tr>
<tr>
<td>Kurtosis</td>
<td>-0.1797</td>
<td>0.2185</td>
<td>-0.6515</td>
</tr>
<tr>
<td>Shapiro-Wilk Test p</td>
<td>0.0685</td>
<td>0.3822</td>
<td>0.0017</td>
</tr>
</tbody>
</table>

The investigation of the kernel density plots – Figure 3(a) for Correctness and Figure 4(a) 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 (last row in Table 1)
Table 2: Descriptive Statistics per Group of Dependent Variable **Duration**

<table>
<thead>
<tr>
<th></th>
<th>Interfaces</th>
<th>Mixins</th>
<th>Traits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Observations</strong></td>
<td>35</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td><strong>Mean [s]</strong></td>
<td>2833.00</td>
<td>2753.33</td>
<td>2856.97</td>
</tr>
<tr>
<td><strong>Standard deviation [s]</strong></td>
<td>718.33</td>
<td>702.84</td>
<td>815.27</td>
</tr>
<tr>
<td><strong>Median [s]</strong></td>
<td>3001.00</td>
<td>2723.00</td>
<td>2636.00</td>
</tr>
<tr>
<td><strong>Median abs. deviation [s]</strong></td>
<td>762.06</td>
<td>612.31</td>
<td>728.70</td>
</tr>
<tr>
<td><strong>Minimum [s]</strong></td>
<td>1244.00</td>
<td>1011.00</td>
<td>1312.00</td>
</tr>
<tr>
<td><strong>Maximum [s]</strong></td>
<td>4102.00</td>
<td>4256.00</td>
<td>4838.00</td>
</tr>
<tr>
<td><strong>Skew [1]</strong></td>
<td>-0.3657</td>
<td>-0.0757</td>
<td>0.5375</td>
</tr>
<tr>
<td><strong>Kurtosis [1]</strong></td>
<td>-0.7073</td>
<td>-0.0528</td>
<td>-0.1457</td>
</tr>
<tr>
<td><strong>Shapiro-Wilk Test</strong></td>
<td>0.4215</td>
<td>0.9737</td>
<td>0.4259</td>
</tr>
</tbody>
</table>

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

(b) Box Plot of **Duration**

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

Figure 4: Descriptive Plots per Group of the Dependent Variable **Duration**

and Table 2 and only the **Traits** group for the dependent variable **Correctness** shows a significant ($p \leq 0.05$) difference to the normal distribution, which would make MANOVA not suitable to be applied to **Correctness**. To finally conclude that the **MANOVA** method cannot be applied, we visually inspected the nor-
Table 3: Hypothesis Tests per Group Combination of the Dependent Variable Correctness

<table>
<thead>
<tr>
<th>Interfaces vs. Mixins</th>
<th>Interfaces vs. Traits</th>
<th>Mixins vs. Traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cliff’s δ</td>
<td>-0.4003</td>
<td>0.1361</td>
</tr>
<tr>
<td>sδ</td>
<td>0.1294</td>
<td>0.1434</td>
</tr>
<tr>
<td>vδ</td>
<td>0.0168</td>
<td>0.0206</td>
</tr>
<tr>
<td>zδ</td>
<td>-3.0931</td>
<td>0.9492</td>
</tr>
<tr>
<td>CI&lt;sub&gt;low&lt;/sub&gt;</td>
<td>-0.6212</td>
<td>-0.1496</td>
</tr>
<tr>
<td>CI&lt;sub&gt;high&lt;/sub&gt;</td>
<td>-0.1205</td>
<td>0.4009</td>
</tr>
<tr>
<td>P(X &gt; Y)</td>
<td>0.6985</td>
<td>0.0059</td>
</tr>
<tr>
<td>P(X = Y)</td>
<td>0.0033</td>
<td>0.0134</td>
</tr>
<tr>
<td>P(X &lt; Y)</td>
<td>0.2982</td>
<td>0.5613</td>
</tr>
<tr>
<td>p</td>
<td>0.0029</td>
<td>0.3460</td>
</tr>
<tr>
<td>p&lt;sub&gt;FDR&lt;/sub&gt;</td>
<td>0.0172</td>
<td>0.6668</td>
</tr>
<tr>
<td>Wilcoxon Test W</td>
<td>857</td>
<td>514</td>
</tr>
<tr>
<td>(two-tail, ≠) p&lt;sub&gt;W&lt;/sub&gt;</td>
<td>0.0041</td>
<td>0.3338</td>
</tr>
<tr>
<td>p&lt;sub&gt;W,FDR&lt;/sub&gt;</td>
<td>0.0246</td>
<td>0.6647</td>
</tr>
</tbody>
</table>

We conducted Q-Q plots for both dependent variables, which are depicted in Figure 3c for Correctness and Figure 4c for Duration. All distribution plots indicate 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. [59], we cannot use the Kruskal-Wallis test [70] because it is strongly affected by unequal variances. Therefore, we select a robust non-parametric test called Cliff’s δ [71]. This testing method is unaffected by non-normal data, change in distribution, and (possible) unstable variance.

The results of the Cliff’s δ test are shown in Table 3 for the dependent variable Correctness and in Table 4 for the dependent variable Duration. Due to the fact that we applied this hypothesis test six 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 [72], which suggests to lower the current significance level \( \alpha = 0.05 \) divided by the times a certain test was applied \( (n = 6) \), which would result into \( \alpha' = \frac{\alpha}{n} = \frac{0.05}{6} = 0.0083 \). 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 [73].

According to the FDR adjusted p-values (p<sub>FDR</sub>) in Table 3 and Table 4, there is evidence not to reject some null hypotheses of this study (see Section 3.5). Since Cliff’s δ test is closely related to the Wilcoxon rank sum test [74] (also know as Mann-Whitney test [75]), we performed a two-tailed (p<sub>W</sub>) sample Wilcoxon test for all language construct (group) combinations to determine the possibility of misinterpretations of the used Cliff’s δ test. The results are...
Table 4: Hypothesis Tests per Group Combination of the Dependent Variable Duration

<table>
<thead>
<tr>
<th></th>
<th>Interfaces vs. Mixins</th>
<th>Interfaces vs. Traits</th>
<th>Mixins vs. Traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cliff’s δ</td>
<td>-0.1091</td>
<td>-0.0286</td>
<td>0.0285</td>
</tr>
<tr>
<td>s_δ</td>
<td>0.1418</td>
<td>0.1416</td>
<td>0.1431</td>
</tr>
<tr>
<td>v_δ</td>
<td>0.0201</td>
<td>0.0201</td>
<td>0.0205</td>
</tr>
<tr>
<td>z_δ</td>
<td>-0.7692</td>
<td>-0.2017</td>
<td>0.1993</td>
</tr>
<tr>
<td>CI_low</td>
<td>-0.3734</td>
<td>-0.2985</td>
<td>-0.2484</td>
</tr>
<tr>
<td>CI_high</td>
<td>0.1716</td>
<td>0.2456</td>
<td>0.3011</td>
</tr>
<tr>
<td>P(X &gt; Y)</td>
<td>0.5541</td>
<td>0.5143</td>
<td>0.4857</td>
</tr>
<tr>
<td>P(X = Y)</td>
<td>0.0009</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>P(X &lt; Y)</td>
<td>0.4450</td>
<td>0.4857</td>
<td>0.5143</td>
</tr>
<tr>
<td>p</td>
<td>0.4445</td>
<td>0.8407</td>
<td>0.8426</td>
</tr>
<tr>
<td>p_FDR</td>
<td>0.6668</td>
<td>0.8426</td>
<td>0.8426</td>
</tr>
</tbody>
</table>

Wilcoxon Test W: 640.5 612 545

Presented at the bottom of Table 3 and Table 4 along with the appropriate FDR adjusted p-value p_{FDR}.

Only for the Correctness of Interfaces vs. Mixins we found evidence of a better understanding of answering structural, behavioral, and operational questions about given ASM specifications.

The test results on Correctness are significant for the comparison of the language constructs Interfaces and Mixins. This would suggest to reject $H_{0,1}$ and to accept $H_{A,1}$. Nevertheless, the hypothesis test results on the dependent variable Duration are not significant which would indicate not to reject $H_{0,1}$.

For the inferential statistical test results on Correctness and Duration we can observe that those dependent variables do not show any significant difference for the comparison of Mixins vs. Traits as well as for the comparison of Interfaces vs. Traits, which suggests not to reject the null hypotheses $H_{0,2}$ and $H_{0,3}$. Therefore, both alternative hypotheses $H_{A,2}$ and $H_{A,3}$ cannot be accepted in this controlled experiment.

6. Discussion

The descriptive statistics are not in favor of any language construct in the overall comparison. By looking only at the Correctness, Interfaces and Traits seem to perform better than Mixins.

The median of the Correctness variable is for language construct Interfaces 54%, Mixins 47%, and Traits 52%, which can be considered rather low in an overall participants’ correctness performance. Due to the fact that all participants have no prior knowledge of ASMs and formal methods in general (checked by an informational question in the survey), a median for the correctness between 47% to 54% can be considered a rather good result in this study. For the Duration descriptive statistical results, Traits and Mixins seem
Table 5: Correlation per Group of the Depended Variables Correctness to Duration

<table>
<thead>
<tr>
<th></th>
<th>Interfaces</th>
<th>Mixins</th>
<th>Traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spearman’s ρ</td>
<td>0.1720</td>
<td>-0.1277</td>
<td>0.1428</td>
</tr>
<tr>
<td>p</td>
<td>0.3231</td>
<td>0.4788</td>
<td>0.4204</td>
</tr>
<tr>
<td>S</td>
<td>5911.7954</td>
<td>6748.2555</td>
<td>5610.2857</td>
</tr>
</tbody>
</table>

Figure 5: Scatter Plot per Group of the Dependent Variables Correctness to Duration

to perform better than Interfaces. The median of the Duration variable is for language construct Interfaces 3001s (50min 1s), Mixins 2723s (45min 23s), and Traits 2636 (43min 56s), which are good results in the scope of the processed survey and the achieved Correctness results with a limited experiment time of 105min (1h 45min). Note that the highest participant duration was 4838s (1h 20min 38s).

In the inferential statistics Interfaces show a significantly better understanding than Mixins in terms of Correctness. If we compare all language constructs, there is no real difference in terms of understanding for the inferential statistics. This implies that for the ASM language user (novice and moderately advanced software architect, designer, or developer) it does not matter, which language construct is used.

By looking at the scatter plot (Figure 5) and correlation (Table 5) of the two dependent variables Correctness and Duration, we cannot observe a linear trend that the dependent variables are correlated since in all language constructs the significance p-value is greater than the significance level of α < 0.5. The kernel density plots for the participants’ self assessment is depicted in Figure 6. The self assessment was measured by calculating the difference between the actual Correctness value and the participants Confidence value that a certain task was correct. A self assessment value ≤ 0 means overestimated and ≥ 0 means underestimated the Correctness of the given experiment answers. All three groups show a similar self assessment with its peak in the underestimated section. This implies that all three language constructs show a similar participants’ self assessment regarding their Confidence in the Correctness of their given solutions.
6.1. 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 105 minutes of the experiment, the chances for maturation effects and experimental fatigue were limited, and we did not observe such. Furthermore, due to the randomized design of the experiment every participant is only tested once with one assigned treatment – interfaces, mixins, or traits – to carry out the experiment for the provided tasks. Therefore, 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. This rules 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).

6.2. 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. As only one participant has prior knowledge in Rust and Scala language, only further seven participants have prior knowledge in Scala, but all participants know Java, a higher familiarity with Interfaces than with the other two tested language constructs can be assumed in our participants. Nonetheless, in our study results, the understandability of Traits is almost equal to the understandability of Interfaces, which might be surprising. Further study is needed to investigate if the relation between the two language constructs – Interfaces and Traits – is different for developers highly familiar with Traits.

In addition to the types of the participants in this experiment (students as novice and moderately advanced software architect, designer, or developer), 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 patterns for distributed systems.

In order to reduce the risk that participants are biased to identify the used language construct in the experiment, we use the syntax keyword feature for all three language constructs under investigation and not the well known abstraction keywords interface, mixin, or trait with are highly familiar to participants in modern programming languages.

6.3. Threats to Construct Validity

We focus in this study on the understandability of language constructs for an ASM language. The understandability is measured by two dependent variables namely correctness and duration. These two dependent variables are commonly used to measure the construct understandability (cf. Hoisl et al. [64], Czepa and Zdun [51]), but it cannot be ruled out that other constructs would be a better measure for understandability.

Berger et al. [76] for example uses the concept of efficiency in their controlled experiment. The construct efficiency measures the ratio of correct answers to time. In this case the amount of time represents only the time it takes after receiving the stimuli to answer certain questions. Since we allow in this controlled experiment the participant to reread the stimuli if needed multiple times during the processing of the questions, the amount of time includes, besides the actual time to answering questions, the time of comprehending the task stimuli, which compromises to reason about efficiency. In another study [54] we established by the controlled experiment design that the participants track the timings (duration) of comprehending and answering separately which allows to reason about efficiency.

6.4. Threats to Content Validity

In this study, we only focus on three language constructs – interfaces, mixins, and traits. The understandability is tested for three ASM syntax variations, not commonly existing in today’s languages and tools, which use one of the language constructs. Testing more complex scenarios (more structures and language constructs) would improve the content validity.

6.5. Threats to Conclusion Validity

Due to some missing timestamps for the dependent variable duration and missing answers 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 ASM specifications in a certain language construct are too complex or not understood at all. Deleting those would compromise the conclusion validity. To improve the conclusion validity, we selected a test with great statistical power which fits the best explored model assumptions of all statistical tests suitable for the collected data set.
6.6. Inferences

Based on the evidence found in this research, a possible use of either Inter-
faces and Traits in ASMs language designs should provide a similar under-
standability. As Mixins perform significantly worse for the dependent variable
Correctness than Interfaces, they should be used with more caution and might
perform worse in some respects than the other two language constructs. Re-
garding the dependent variable Duration, it seems that for all the different
kinds of textual language construct representations the participants need a sim-
ilar duration to process the surveys and without further studies no generalized
claim can be drawn from the gathered results.

6.7. Relevance to Practice

State-of-the-art abstractions are key for acceptance of formal methods in
practice. So far many formal specification languages lack in their support for
other advanced language constructs, such as Interfaces, Mixins, 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 language engineers in
practice to choose, specify, and implement new language constructs in existing
or newly developed programming/specification languages in order to achieve a
more understandable language syntax for the language user.

Many formalisms, including ASMs have been implemented in different pro-
gramming and/or specification languages. Our empirical results can help lan-
guage users of these formalisms to choose one of those languages using the avail-
able 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.

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

7. Conclusion

This article reports on a controlled experiment with 105 participants on
the understandability of language constructs tested for the applicability in the
context of an ASMs modeling language as a representative for other ASM-
based languages and other state-based formal methods.

The focus of the study is the understanding of structural and behavioral
properties of given ASMs specifications modeled in three ASMs language syntax
extensions, which are not yet part of ASMs or any other ASM-based language,
namely Interfaces, Mixins, and Traits.
According to the descriptive and inferential statistics, *Interfaces* and *Traits* can be used interchangeably with regard to their expectations in terms of understandability, whereas *Mixins* should be used with caution, as they show significantly worse understanding in comparison with *Interfaces* for the dependent variable **Correctness**. As *Mixins* show no significant difference in terms of **Duration** compared to *Interfaces* and for both dependent variables compared to *Traits*, more research is needed to understand the reasons why they perform worse with regard to only one dependent variable.

This study is a first step towards establishing an understandable language design with regard to language constructs for structuring behavioral specifications. The outcomes can be used by language designers and compiler engineers to define a suitable language construct in an ASM language like **CASM**. They indicate that at least some of the heated debates on language constructs can be neglected and the best suited abstraction in the context of other language design concerns like language consistency can be chosen. It would be interesting to study further if our results can be transferred to other state-based formal methods and maybe even to abstractions in object-oriented languages.

**Acknowledgements**

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

**References**


URL http://www.r-project.org


[75] H. B. Mann, D. R. Whitney, On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other, The annals of mathematical statistics (1947) 50–60.
