The Journal of Systems & Software 178 (2021) 110987

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

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

ARTICLE INFO

Article history:
Received 26 May 2020
Received in revised form 29 January 2021
Accepted 21 April 2021
Available online 27 April 2021

Keywords:
Abstract State Machines
Empirical software engineering
Understandability
Language constructs
Controlled experiment

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 comparing interfaces with mixins.

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 (1995) in the beginning of the 1990s. Several scientists of different research fields used and applied the ASM theory and its ASM method (Börger and Stärk, 2003). This usage ranges from software, hardware and system engineering perspectives to specify, analyze, verify, and validate systems in a formal way (Raschke et al., 2020). The diversity of ASM-based applications ranges from formal specification of semantics of programming languages, such as those for Java by Stärk et al. (2001) or Very High Speed Integrated Circuit Hardware Description Language (VHDL) by Sasaki (1999), compiler back-end verification by Lezuo (2014), software run-time verification by Barnett and Schulte (2001), software and hardware architecture modeling e.g. of Universal Plug and Play (UPnP) by Glässer and Veanes (2002), to even Reduced Instruction Set Computing (RISC) designs by Huggins and Campenhout (1998).

By definition, an ASM formally describes the evolution of function states through dedicated transaction rules in a step-by-step manner[1] and are used to specify sequential, parallel, concurrent, reflective, and even quantum algorithms (Börger and Stärk, 2003). 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 (1995). Additionally, several theory improvements were provided to increase the expressiveness of ASM languages which were summarized by Börger and Stärk (2003) and Börger and Raschke (2018). The landscape of developed ASM languages and the corresponding tools is rather limited nowadays. Best known are the ASM implementations AsmetaL (Gargantini et al., 2008) and CoreASM (Farahbod et al., 2007).

AsmetaL provides a feature-rich tool set to model, analyze, interpret, and generate code of described ASM specifications.[2] The core of AsmetaL is implemented and based on the Eclipse Modeling Framework (EMF) and provides therefore a Java-based interpreter.

☆ Editor: Earl Barr.

* Corresponding author.
E-mail addresses: philipp.paulweber@univie.ac.at (P. Paulweber), georg.simhandl@univie.ac.at (G. Simhandl), uwe.zdun@univie.ac.at (U. Zdun).

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

[2] See https://asmeta.github.io for the AsmetaL project site.

https://doi.org/10.1016/j.jss.2021.110987

CoreASM[3] is another Java-based interpreter implementation for ASM specifications. Its main focus is on the language extensibility which is supported through the adaption of the parser implementation (Farahbod et al., 2007). 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 (Gurevich et al., 2004) and Corinthian Abstract State Machine (CASM) (Lezuo et al., 2013). 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 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. (2013) established compilation techniques to outperform CoreASM and AsmL in terms of ASM execution performance. There are several other ASM language tool implementations like AsmGofer (Schmid, 2001) or eXtensible ASM (XASM) (Anlauff, 2000), 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 (Jackson, 2002), Event-B (Abrial et al., 2010), Temporal Logic of Actions (TLA) (Lamport, 1994), Vienna Development Method (VDM) (Bjørner, 1979), and Z (Potter et al., 1996).

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 (Mernik et al., 2004). While very few have embraced basic object-oriented abstractions such as classes and inheritance, more advanced language constructs are usually missing. Mernik et al. (1998) point out that the lack of such object-oriented abstractions in formal methods is one of main the reason why formal methods and their languages are not widely used and are more or less unpopular compared to feature-rich programming languages. Börger (2018) suggests in one of his latest article 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 (Canning et al., 1989), mixins (Flatt et al., 1998), or traits (Schärli et al., 2003) in addition to classes and inheritance. All of those three language construct have similar and some different properties and characteristics, which are depicted in Fig. 1 and described as follows:

Interfaces define (typed) operations (signatures) to which an implementer of a certain interface (type) has to conform (Canning et al., 1989). Therefore, an interface defines a so called *contract* (Meyer, 1992). 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 (Flatt et al., 1998; Bracha and Cook, 1990). 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 (Schärli et al., 2003). 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.

Fig. 1. Overview of language construct properties.

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 (Odersky et al., 2008), which offers a trait syntax that is similar to the Java 8 (Potts and Friedel, 2018) 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 (Matsakis and Klock II, 2014), 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 (Lezuo et al., 2013, 2014; Paulweber and Zdun, 2016; Paulweber et al., 2018) 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 (Van Solingen et al., 2002) 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 University of Vienna.[4]

Our hypotheses are influenced by the debate in the object-oriented community, which recently discuss traits often more favorably than mixins.[5] In particular, mixins contain state information whereas traits do not, mixins use implicit conflict resolution whereas traits use explicit resolution and mixins are

[3] See https://github.com/CoreASM for the CoreASM project site.

[4] See https://cs.univie.ac.at for Faculty of Computer Science website.

[5] See, e.g. https://stackoverflow.com/questions/925609.

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[6] or point out that "Traits are Interfaces"[7] 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 (Gurevich and Tillmann, 2001), 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 (Börger and Stärk, 2003).

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 (Börger and Raschke, 2018) 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 (Gurevich, 2000), 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* (Börger and Raschke, 2018). 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 (2018).

2.2. ASM language representative

In this study, we use the basic syntax elements from the CASM language[8] (Paulweber et al., 2018). 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. 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.

6 See, e.g. https://stackoverflow.com/questions/9205083.

7 See, e.g. https://blog.rust-lang.org/2015/05/11/traits.html.

8 See https://casm-lang.org/syntax for CASM language description.

```
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

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

Listing 3: Named *Rule* Definition Example

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 (2018). 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 (Gurevich et al., 2004), 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 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*.

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 (Liskov and Zilles, 1974; Canning et al., 1989) 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.[9] This syntax is primarily influenced by the Java programming language (Potts and Friedel, 2018) interface syntax.

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 (Moon, 1986; Flatt et al., 1998). 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.[10] This syntax is primarily influenced by the Scala programming language (Odersky et al., 2008) 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.

[9] See `form_ifaces.pdf` at Paulweber et al. (2021b).
[10] See `form_mixins.pdf` at Paulweber et al. (2021b).

```
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 4: *Interfaces*-Based Example Specification

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

implement 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

  // overwrite of 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
}

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

Listing 6: *Traits*-Based Example Specification

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.[11] This `feature` and `implement` syntax is influenced by the Rust programming language (Matsakis and Klock II, 2014) trait syntax.[12]

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 (Murphy-Hill et al., 2005). Apel and Batory present a case study comparing aspect and feature abstractions using a mixin layer approach to unify the two (Apel and Batory, 2006). Batory et al. present another case study on achieving extensibility through product-lines and domain-specific languages using a mixin-based approach (Batory et al., 2002). 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 (Clarke and Wing, 1996; De Alfaro and Henzinger, 2001; Cheon et al., 2005) and architecture description languages that offer formal representations (Oquendo, 2004; Garlan, 2003). Traits and mixins, 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 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 (Sobel and Clarkson, 2002). Czepa and Zdun (2018) and Czepa et al. (2017) have studied the understandability of formal methods for temporal property specification using similar research methods as used in this study.

Ferrarotti et al. (2020) report on a recent study where ASM-based high-level software specifications are extracted from Java programs by using a semi-automated approach. This study is of interest, because it maps the Java object-oriented programming language concepts to the ASM sub-machine (Ferrarotti et al., 2020) concept in order to represent the abstract type (interfaces) and sub-classing mechanisms.

Related to this study, we conducted another controlled experiment (Paulweber et al., 2021a) 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 (Paulweber et al., 2021a) 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. (2008) on how empirical research shall be conducted and reported in software engineering. Moreover, the guidelines by Kitchenham et al. (2002), Wohlin et al. (2012), and Juristo and Moreno (2013) 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. (2017).

11 See `form_traits.pdf` at Paulweber et al. (2021b).

12 See https://doc.rust-lang.org/rust-by-example/trait.html for the discussion.

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.

ASE: 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) (Fowler, 2010) and Model-Driven Development (MDD) (Beydeda et al., 2005) 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) a participant answered the survey determined the bonus points achieved (for *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 ASM specification language, see Section 2). Due to random assignment of the participants to groups – *Interfaces* (Group A), *Mixins* (Group C), and *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 (DSE and 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. (2002) 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 Fig. 2c, which summarizes the participants' industrial work experiences).

Other studies by Svahnberg et al. (2008) or Salman et al. (2015) 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 *Message Queue*, *Publish–Subscribe*, and *Remote Procedure Call* patterns as example applications inspired by examples provided by Börger and Raschke (2018).

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

(1) Information Sheet An experiment information document[15] explaining the ASM language syntax and semantics **without the experiments' language construct** syntax and semantics extensions.

(2) Survey Form Three experiment survey forms[16] 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**.

[13] See https://ufind.univie.ac.at/en/course.html?lv=053020&semester=2018S.

[14] See https://ufind.univie.ac.at/en/course.html?lv=052500&semester=2018S.

[15] See `info.pdf` at Paulweber et al. (2021b).

[16] See `form_ifaces.pdf`, `form_mixins.pdf`, and `form_traits.pdf` at Paulweber et al. (2021b).

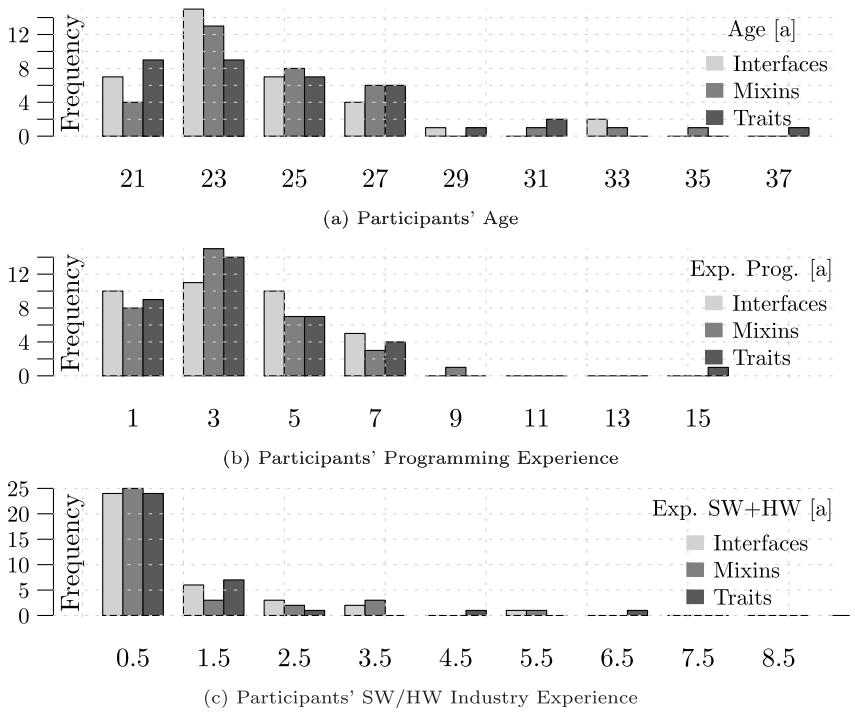


Fig. 2. Histograms per group of participants' background information.

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.

These two dependent variables are commonly used to measure the construct **understandability** (cf. Hoisl et al., 2014; Czepa and Zdun, 2018). 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 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[18] OpenDocument Spreadsheet (ODS) file (Parsons, 2012). 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 Sections 4.1 and 4.2. We did not observe any unforeseen difficulties and did not deviate from the experiment plan.

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

[18] See https://www.libreoffice.org for version `6.1.4.2`.

5. Analysis

All statistical analysis was performed with the software tool R.[19] The analysis processes[20] 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]

5.1. Data-set preparation

The raw data[22] collected during the experiment execution phase (see Section 4) was prepared[23] in the following manner: (1) the obtained LibreOffice ODS file (Parsons, 2012) was exported to a Comma-Seperated Values (CSV) file (Shafranovich, 2005); (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 $C = \sum_{1}^{n=3} \frac{C_n * n}{6}$, 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[24]; (5) and stored as an R Data-Set (RDS) file (R Development Core Team, 2008) for further processing and analysis.

5.2. Descriptive statistics

The participants' experience and characteristics (background information) are captured in the experiment by: age (see Fig. 2a), gender, course, and level of education, programming experience (see Fig. 2b), modeling experience, software (SW) and hardware (HW) industry experience (see Fig. 2c), and programming and specification languages used.[25] Overall, the random distribution of the participants to the experiment groups is almost balanced.

The participants' programming experience (see Fig. 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**[26] and this acquired data is visualized as a kernel density plot in Fig. 3a and a box plot in Fig. 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

[19] See https://www.r-project.org for version `3.5.2`.

[20] See `analyze.r` at Paulweber et al. (2021b).

[21] See `install.r` at Paulweber et al. (2021b).

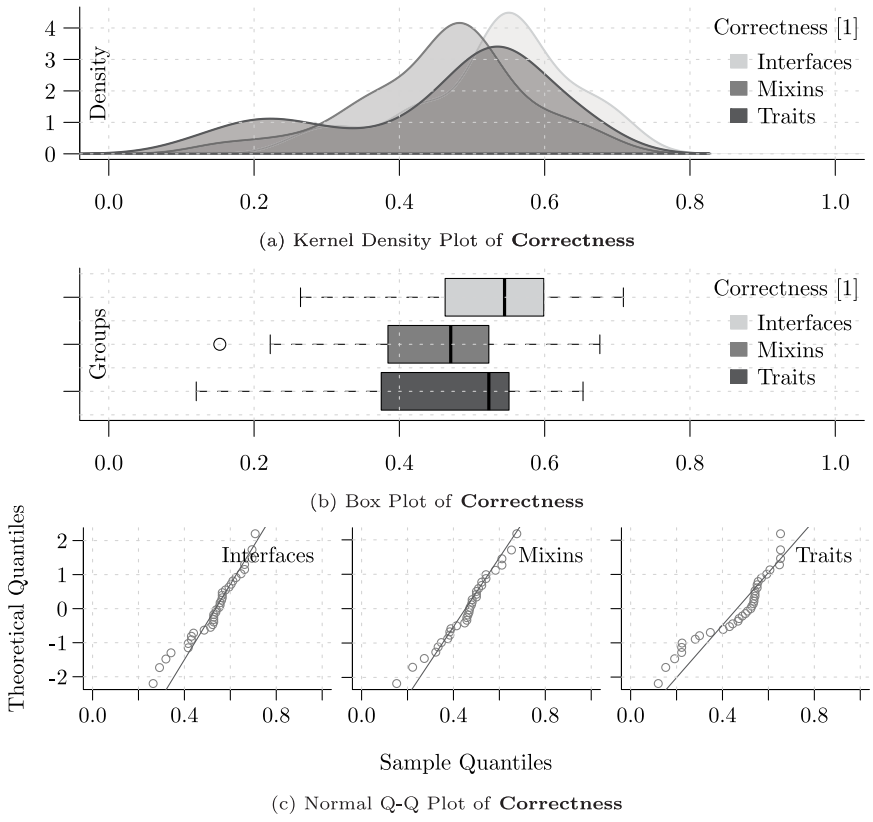
[22] The data-set is published in the long term open data archive Zenodo (Paulweber et al., 2021b) together with all documents and R scripts.

[23] See `prepare.r` at (Paulweber et al., 2021b).

[24] For detailed formula, see `prepare.r` Line 235–340 at Paulweber et al. (2021b).

[25] See `appendix.pdf` at Paulweber et al. (2021b) for supplementary background information.

[26] Unit is correctness rate between 0.0 and 1.0 (denoted [1]).



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

(b) Box Plot of **Correctness**

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

Fig. 3. Descriptive plots per group of the dependent variable **Correctness**.

Table 1
Descriptive statistics per group of dependent variable **correctness**.

	Interfaces	Mixins	Traits
Observations [1]	36	34	35
Mean [1]	0.5294	0.4574	0.4598
Standard deviation [1]	0.1110	0.1147	0.1533
Median [1]	0.5448	0.4707	0.5231
Median abs. deviation [1]	0.0869	0.0950	0.1030
Minimum [1]	0.2639	0.1528	0.1204
Maximum [1]	0.7083	0.6759	0.6528
Skew [1]	−0.6132	−0.4984	−0.7713
Kurtosis [1]	−0.1797	0.2185	−0.6515
Shapiro–Wilk Test p [1]	0.0685	0.3822	0.0017

Table 2
Descriptive statistics per group of dependent variable **duration**.

	Interfaces	Mixins	Traits
Observations [1]	35	33	34
Mean [s]	2833.00	2753.33	2856.97
Standard deviation [s]	718.33	702.84	815.27
Median [s]	3001.00	2723.00	2636.00
Median abs. deviation [s]	762.06	612.31	728.70
Minimum [s]	1244.00	1011.00	1312.00
Maximum [s]	4102.00	4256.00	4838.00
Skew [1]	−0.3657	−0.0757	0.5375
Kurtosis [1]	−0.7073	−0.0528	−0.1457
Shapiro–Wilk Test p [1]	0.4215	0.9737	0.4259

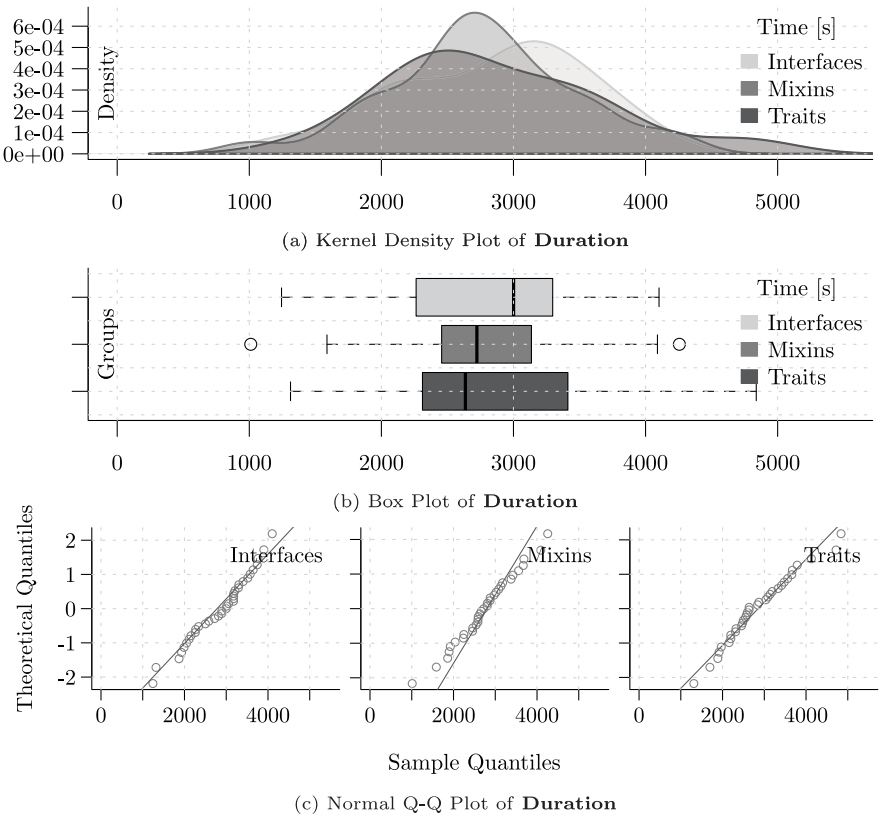
this distribution is strongly right skewed. According to the kernel density plot, the data does not appear to be normally distributed, and all three distributions look different, which implies unequal variances. The *Interfaces* has its peak at 0.55 and *Mixins* has its peak at 0.45. In contrast to the two other groups, the *Traits* groups has two peaks, one at about 0.215 and the other one at about 0.525.

Table 2 contains the number of observations, central tendency measures, and dispersion measures per language construct for the dependent variable **Duration**[27] and this acquired data is visualized as a kernel density plot in Fig. 4a and a box plot in Fig. 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) (Borgen and Seling, 1978) would be a suitable statistical procedure, but necessary assumptions must be met to apply this method. The investigation of the kernel density plots – Fig. 3a for **Correctness** and Fig. 4a 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 (Shapiro and Wilk, 1965) (last row in Tables 1 and 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

[27] Unit is duration in seconds (denoted [s]).



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

(b) Box Plot of **Duration**

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

Fig. 4. Descriptive plots per group of the dependent variable **Duration**.

Table 3
Hypothesis tests per group combination of the dependent variable **correctness**.

	Interfaces vs. Mixins	Interfaces vs. Traits	Mixins vs. Traits
Cliff's δ	−0.4003	−0.2667	0.1361
s_δ	0.1294	0.1317	0.1434
v_δ	0.0168	0.0173	0.0206
z_δ	−3.0931	−2.0254	0.9492
CI_{low}	−0.6212	−0.5023	−0.1496
CI_{high}	−0.1205	0.0059	0.4009
$P(X > Y)$	0.6985	0.6294	0.4252
$P(X = Y)$	0.0033	0.0079	0.0134
$P(X < Y)$	0.2982	0.3627	0.5613
p	0.0029	0.0467	0.3460
p_{FDR}	**0.0172**	0.1401	0.6668
Wilcoxon Test W	857	798	514
(two-tail, $\neq$) p_W	0.0041	0.0540	0.3338
$p_{W_{FDR}}$	**0.0246**	0.1620	0.6647

Table 4
Hypothesis tests per group combination of the dependent variable **duration**.

	Interfaces vs. Mixins	Interfaces vs. Traits	Mixins vs. Traits
Cliff's δ	−0.1091	−0.0286	0.0285
s_δ	0.1418	0.1416	0.1431
v_δ	0.0201	0.0201	0.0205
z_δ	−0.7692	−0.2017	0.1993
CI_{low}	−0.3734	−0.2985	−0.2484
CI_{high}	0.1716	0.2456	0.3011
$P(X > Y)$	0.5541	0.5143	0.4857
$P(X = Y)$	0.0009	0.0000	0.0000
$P(X < Y)$	0.4450	0.4857	0.5143
p	0.4445	0.8407	0.8426
p_{FDR}	0.6668	0.8426	0.8426
Wilcoxon Test W	640.5	612	545
(two-tail, $\neq$) p_W	0.4431	0.8430	0.8459
$p_{W_{FDR}}$	0.6647	0.8459	0.8459

make MANOVA not suitable to be applied to **Correctness**. 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 Fig. 3c for **Correctness** and Fig. 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. (2017), we cannot use the Kruskal–Wallis test (Kruskal and Wallis, 1952) because it is strongly affected by unequal variances. Therefore, we select a robust non-parametric test called Cliff's δ (Cliff, 1993). 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 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 (Dunn, 1958), 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 $\alpha' = \frac{\alpha}{n} = \frac{0.05}{6} = 0.008\dot{3}$. 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 (Benjamini and Hochberg, 1995). According to the FDR adjusted p-values (p_{FDR}) in Tables 3 and 4, there

is evidence not to reject some null hypotheses of this study (see). Since Cliff's δ test is closely related to the Wilcoxon rank sum test (Wilcoxon, 1945) (also know as Mann–Whitney test Mann and Whitney, 1947), we performed a two-tailed (p_W) 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 presented at the bottom of Tables 3 and 4 along with the appropriate FDR adjusted p-value $p_{W_{\text{FDR}}}$.

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 $\mathbf{H_{0,1}}$ and to accept $\mathbf{H_{A,1}}$. Nevertheless, the hypothesis test results on the dependent variable **Duration** are not significant which would indicate not to reject $\mathbf{H_{0,1}}$. 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 $\mathbf{H_{0,2}}$ and $\mathbf{H_{0,3}}$. Therefore, both alternative hypotheses $\mathbf{H_{A,2}}$ and $\mathbf{H_{A,3}}$ 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 to perform better than *Interfaces*. The median of the **Duration** variable is for language construct *Interfaces* 3001*s* (50*min* 1*s*), *Mixins* 2723*s* (45*min* 23*s*), and *Traits* 2636 (43*min* 56*s*), which are good results in the scope of the processed survey and the achieved **Correctness** results with a limited experiment time of 105 min (1*h* 45*min*). Note that the highest participant duration was 4838*s* (1*h* 20*min* 38*s*).

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 (Fig. 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 $\alpha < 0.5$. The kernel density plots for the participants' *self assessment* is depicted in Fig. 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.

Table 5
Correlation per group of the depended variables **correctness** to **duration**.

	Interfaces	Mixins	Traits
Spearman's ρ	0.1720	−0.1277	0.1428
p	0.3231	0.4788	0.4204
S	5911.7954	6748.2555	5610.2857

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

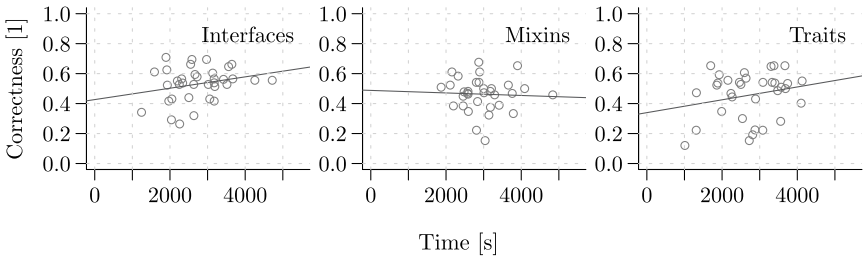


Fig. 5. Scatter plot per group of the dependent variables **correctness** to **Duration**.

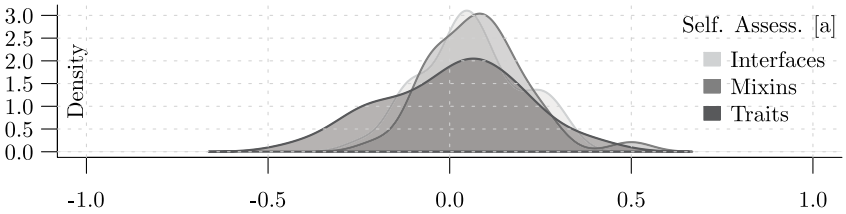


Fig. 6. Kernel density plot per group of participants' *Self Assessment*.

measure the construct understandability (cf. Hoisl et al., 2014; Czepa and Zdun, 2018), but it cannot be ruled out that other constructs would be a better measure for understandability.

Berger et al. (2016) 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 (Paulweber et al., 2021a) 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 *Interfaces* and *Traits* in ASM language designs should provide a similar understandability. 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. Regarding the dependent variable **Duration**, it seems that for all the different kinds of textual language construct representations the participants need a similar 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* (Kleppe, 2009) 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 programming and/or specification languages. Our empirical results can help *language users* of these formalisms 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.

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 ASM-based 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 ASM specifications modeled in three CASM language syntax extensions, which are not yet part of CASM 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 ASM 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.

CRediT authorship contribution statement

Philipp Paulweber: Conceptualization, Methodology, Software, Formal analysis, Visualization, Investigation, Data curation, Resources, Writing - original draft, Writing - review & editing. **Georg Simhandl:** Conceptualization, Data curation, Validation, Writing - review & editing. **Uwe Zdun:** Supervision, Conceptualization, Methodology, Writing - original draft, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

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 language abstractions.

References

Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L., 2010. Rodin: an open toolset for modelling and reasoning in event-b. Int. J. Softw. Tools Technol. Transf. 12 (6), 447–466.

Anlauff, M., 2000. XASM – An extensible, component-based abstract state machines language. In: Abstract State Machines-Theory and Applications. Springer, pp. 69–90.

Apel, S., Batory, D., 2006. When to use features and aspects?: A case study. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering. GPCE '06, ACM, New York, NY, USA, pp. 59–68. http://dx.doi.org/10.1145/1173706.1173716, URL http://doi.acm.org/10.1145/1173706.1173716.

Barnett, M., Schulte, W., 2001. Spying on components: a runtime verification technique, in: Proceedings of the Workshop on Specification and Verification of Component-Based Systems. SAVCBS'01, 2001, pp. 7–13.

Batory, D., Johnson, C., MacDonald, B., von Heeder, D., 2002. Achieving extensibility through product-lines and domain-specific languages: A case study. ACM Trans. Softw. Eng. Methodol. 11 (2), 191–214. http://dx.doi.org/10.1145/505145.505147, URL http://doi.acm.org/10.1145/505145.505147.

Benjamini, Y., Hochberg, Y., 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. J. R. Stat. Soc. Ser. B Methodol. 289–300.

Berger, T., Völter, M., Jensen, H.P., Dangprasert, T., Siegmund, J., 2016. Efficiency of projectional editing: A controlled experiment, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 763–774.

Beydeda, S., Book, M., Gruhn, V., et al., 2005. Model-Driven Software Development, Vol. 15. Springer.

Bjørner, D., 1979. The vienna development method (vdm). In: Mathematical Studies of Information Processing. Springer, pp. 326–359.

Borgen, F.H., Seling, M.J., 1978. Uses of discriminant analysis following manova: Multivariate statistics for multivariate purposes. J. Appl. Psychol. 63 (6), 689.

Börger, E., 2018. Why programming must be supported by modeling and how. In: International Symposium on Leveraging Applications of Formal Methods. Springer, pp. 89–110.

Börger, E., Raschke, A., 2018. Modeling Companion for Software Practitioners. Springer.

Börger, E., Stärk, R., 2003. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer Science & Business Media.

Bracha, G., Cook, W., 1990. Mixin-based inheritance. ACM SIGPLAN Not. 25 (10), 303–311.

Canning, P.S., Cook, W.R., Hill, W.L., Olthoff, W.G., 1989. Interfaces for strongly-typed object-oriented programming. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications. OOPSLA '89, ACM, New York, NY, USA, pp. 457–467. http://dx.doi.org/10.1145/74877.74924, URL http://doi.acm.org/10.1145/74877.74924.

Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S., 2005. Model variables: Cleanly supporting abstraction in design by contract. Softw. - Pract. Exp. 35 (6), 583–599.

Clarke, E.M., Wing, J.M., 1996. Formal methods: State of the art and future directions. ACM Comput. Surv. 28 (4), 626–643. http://dx.doi.org/10.1145/242223.242257, URL http://doi.acm.org/10.1145/242223.242257.

Cliff, N., 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. Psychol. Bull. 114 (3), 494.

Czepa, C., Tran, H., Zdun, U., Kim, T.T.T., Weiss, E., Ruhsam, C., 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, pp. 155–164.

Czepa, C., Zdun, U., 2018. On the understandability of temporal properties formalized in linear temporal logic, property specification patterns and event processing language. IEEE Trans. Softw. Eng.

De Alfaro, L., Henzinger, T.A., 2001. Interface theories for component-based design. In: International Workshop on Embedded Software. Springer, pp. 148–165.

Dunn, O.J., 1958. Estimation of the means of dependent variables. Ann. Math. Stat. 1095–1111.

Farahbod, R., Gervasi, V., Glässer, U., 2007. CoreASM: An extensible ASM execution engine. Fund. Inform. 77 (1–2), 71–104.

Ferrarotti, F., Moser, M., Pichler, J., 2020. Stepwise abstraction of high-level system specifications from source code. J. Comput. Lang. 60, 100996.

Flatt, M., Krishnamurthi, S., Felleisen, M., 1998. Classes and mixins. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '98, ACM, New York, NY, USA, pp. 171–183. http://dx.doi.org/10.1145/268946.268961, URL http://doi.acm.org/10.1145/268946.268961.

Fowler, M., 2010. Domain-Specific Languages. Pearson Education.

Gargantini, A., Riccobene, E., Scandurra, P., 2008. A metamodel-based language and a simulation engine for abstract state machines. J. UCS 14 (12), 1949–1983.

Garlan, D., 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, pp. 1–24.

Glässer, U., Veanes, M., 2002. Universal plug and play machine models. In: Design and Analysis of Distributed Embedded Systems. Springer, pp. 21–30.

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

Gurevich, Y., 2000. Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Logic 1 (1), 77–111.
Gurevich, Y., Rossman, B., Schulte, W., 2004. Semantic essence of AsmL. In: Formal Methods for Components and Objects. Springer, pp. 240–259.
Gurevich, Y., Tillmann, N., 2001. Partial updates: Exploration. J. UCS 7 (11), 917–951.
Hoisl, B., Sobernig, S., Strembeck, M., 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, pp. 180–189.
Huggins, J.K., Campenhout, D.V., 1998. Specification and verification of pipelining in the arm2 risc microprocessor. ACM Trans. Des. Autom. Electron. Syst. 3 (4), 563–580.
Jackson, D., 2002. Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11 (2), 256–290.
Jedlitschka, A., Ciolkowski, M., Pfahl, D., 2008. Reporting experiments in software engineering. In: Guide to Advanced Empirical Software Engineering. Springer, pp. 201–228.
Juristo, N., Moreno, A.M., 2013. Basics of Software Engineering Experimentation. Springer Science & Business Media.
Kitchenham, B., Madeyski, L., Budgen, D., Keung, J., Brereton, P., Charters, S., Gibbs, S., Pohthong, A., 2017. Robust statistical methods for empirical software engineering. Empir. Softw. Eng. 22 (2), 579–630.
Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J., 2002. Preliminary guidelines for empirical research in software engineering. IEEE Trans. Softw. Eng. 28 (8), 721–734.
Kleppe, A.G., 2009. Software Language Engineering: Creating Domain-Specific Languages using Metamodels. Addisson-Wesley.
Kruskal, W.H., Wallis, W.A., 1952. Use of ranks in one-criterion variance analysis. J. Amer. Statist. Assoc. 47 (260), 583–621.
Lamport, L., 1994. The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16 (3), 872–923.
Lezuo, R., 2014. Scalable Translation Validation; Tools, Techniques and Framework (Ph.D. thesis). Techn. Univ., Diss., wien.
Lezuo, R., Barany, G., Krall, A., 2013. CASM: Implementing an abstract state machine based programming language, in: Software Engineering (Workshops). pp. 75–90.
Lezuo, R., Paulweber, P., Krall, A., 2014. CASM - optimized compilation of abstract state machines. In: SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems. LCTES, ACM, pp. 13–22.
Liskov, B., Zilles, S., 1974. Programming with abstract data types. In: Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages. ACM, New York, NY, USA, pp. 50–59. http://dx.doi.org/10.1145/800233.807045, URL http://doi.acm.org/10.1145/800233.807045.
Mann, H.B., Whitney, D.R., 1947. On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. 50–60.
Matsakis, N.D., Klock II, F.S., 2014. The rust language. In: ACM SIGAda Ada Letters, Vol. 34. ACM, pp. 103–104.
Mernik, M., Lenic, M., Avdicausevic, E., Zumer, V., 1998. A reusable object-oriented approach to formal specifications of programming languages. L'Objet 4 (3), 273–306.
Mernik, M., Wu, X., Bryant, B., 2004. Object-oriented language specifications: Current status and future trends, in: ECOOP Workshop: Evolution and Reuse of Language Specifications for DSLs. ERLS.
Meyer, B., 1992. Applying'design by contract'. Computer 25 (10), 40–51.
Moon, D.A., 1986. Object-oriented programming with flavors. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications. OOPSLA '86, ACM, New York, NY, USA, pp. 1–8. http://dx.doi.org/10.1145/28697.28698, URL http://doi.acm.org/10.1145/28697.28698.
Murphy-Hill, E.R., Quitslund, P.J., Black, A.P., 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. OOPSLA '05, ACM, New York, NY, USA, pp. 282–291. http://dx.doi.org/10.1145/1094855.1094963, URL http://doi.acm.org/10.1145/1094855.1094963.
Odersky, M., Spoon, L., Venners, B., 2008. Programming in Scala. Artima Inc.
Oquendo, F., 2004. π-Adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Softw. Eng. Notes 29 (3), 1–14.
Parsons, J.J., 2012. Practical Open Source Office: LibreOffice(TM) and Apache OpenOffice, second ed. Course Technology Press, Boston, MA, United States.
Paulweber, P., Pescosta, E., Zdun, U., 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, In: Lecture Notes in Computer Science, vol. 10817, Springer, pp. 39–54.
Paulweber, P., Simhandl, G., Zdun, U., 2021a. Specifying with interface and trait abstractions in abstract state machines: A controlled experiment, paper to TOSEM (accepted, in publication process).
Paulweber, P., Simhandl, G., Zdun, U., 2021b. On the understandability of language constructs to structure the state and behavior in abstract state machine specifications: A controlled experiment. http://dx.doi.org/10.5281/zenodo.4480316, URL https://doi.org/10.5281/zenodo.4480316.
Paulweber, P., Zdun, U., 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, In: Lecture Notes in Computer Science, vol. 9675, Springer, pp. 250–255.
Potter, B., Till, D., Sinclair, J., 1996. An Introduction to Formal Specification and Z. Prentice Hall PTR.
Potts, A., Friedel, D.H., 2018. Java Programming Language Handbook. Coriolis Group Books.
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, URL http://www.r-project.org.
Raschke, A., Méry, D., Houdek, F., 2020. Rigorous state-based methods. In: Proceedings of 7th International Conference. ABZ 2020, Ulm, Germany, May 27–29, 2020, Springer, p. 8.
Salman, I., Misirli, A.T., Juristo, N., 2015. Are students representatives of professionals in software engineering experiments? In: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, Vol. 1. IEEE, pp. 666–676.
Sasaki, H., 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. DATE '99, ACM, New York, NY, USA.
Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P., 2003. Traits: Composable units of behaviour. In: European Conference on Object-Oriented Programming. Springer, pp. 248–274.
Schmid, J., 2001. Introduction to AsmGofer. http://www.tydo.de/AsmGofer.
Shafranovich, Y., 2005. Common format and MIME type for comma-separated values (CSV) files, RFC 4180, Y. Shafranovich. URL http://tools.ietf.org/html/rfc4180.
Shapiro, S.S., Wilk, M.B., 1965. An analysis of variance test for normality (complete samples). Biometrika 52 (3/4), 591–611.
Sobel, A.E.K., Clarkson, M.R., 2002. Formal methods application: An empirical tale of software development. IEEE Trans. Softw. Eng. 28 (3), 308–320.
Stärk, R.F., Schmid, J., Börger, E., 2001. Java and the Java Virtual Machine: Definition, Verification, Validation. Springer Berlin Heidelberg.
Svahnberg, M., Aurum, A., Wohlin, C., 2008. Using students as subjects-an empirical evaluation. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ACM, pp. 288–290.
Van Solingen, R., Basili, V., Caldiera, G., Rombach, H.D., 2002. Goal question metric (GQM) approach. In: Encyclopedia of Software Engineering.
Wilcoxon, F., 1945. Individual comparisons by ranking methods. Biom. Bull. 1 (6), 80–83, URL http://www.jstor.org/stable/3001968.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer Science & Business Media.

Paulweber, Philipp is a researcher at the Faculty of Computer Science, University of Vienna, Austria. His research areas include state-based formal methods, in particular Abstract State Machines (ASM), compiler construction, model-driven engineering, software and/or hardware architecture and engineering. He received a master degree in computer engineering from the Vienna University of Technology (TU Wien) in 2014. During that time he was employed at the company TTTech as embedded systems engineer and applied industrial standards and tools based on ISO26262 and AUTOSAR. Currently, he is pursuing a Ph.D. in computer science at the University of Vienna, Austria. Philipp has already published more than 4 peerreviewed scientific articles.

Simhandl, Georg is a senior researcher at the Faculty of Computer Science, University of Vienna. Before joining the Research Group Software Architecture in 2017, he founded Adaptivia GmbH in 2008, a company focusing on Intelligent Sensor Networks. He has led numerous research and development projects in the area of wireless sensor networks, adaptive systems and IoT security. Georg holds a Ph.D. in information systems research (2007) from the Vienna University of Business Administration and Economics. His research focuses on software design and architecture at the crossroads of distributed systems engineering, program comprehension, machine learning and cognitive science.

Zdun, Uwe is a full professor for software architecture at the Faculty of Computer Science, University of Vienna, Austria. Before that, he worked as assistant professor at the Vienna University of Technology and the Vienna University of Economics respectively. He received his doctoral degree from the University of Essen in 2002. His research focuses on software design and architecture, distributed systems engineering, and empirical software engineering. Uwe has published more than 210 articles in peer-reviewed journals, conferences, book chapters, and workshops. He has participated in 26 R&D projects. Uwe is editor of the journal TPLoP published by Springer, Associate Editor of the Computing journal published by Springer, and Associate Editor-in-Chief for design and architecture for the IEEE Software magazine.