universität wien

DISSERTATION / DOCTORAL THESIS

Titel der Dissertation / Title of the Doctoral Thesis

„The Evolving of CASM: Modern Compiler Engineering and Empirical Guided Language Design for a Rigorous State-Based Method“

verfasst von / submitted by

Dipl.-Ing. Philipp Paulweber, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Doktor der technischen Wissenschaften (Dr. techn.)

Wien, 2022 / Vienna 2022

Studienkennzahl lt. Studienblatt / Degree programme code as it appears on the student record sheet:	A 786 880
Dissertationsgebiet lt. Studienblatt / Field of study as it appears on the student record sheet:	Informatik / Computer Science
Betreut von / Supervisor:	Univ.-Prof. Dr. Uwe Zdun

For
Habibi
نور

Preface

Corinthian Abstract State Machine
Language, Interpreter, and Compiler

The origin of the name *Corinthian is unclear,*
whether it is taken from "the letters,
the pillars, the leather, the place,
or the mode of behavior".

Puck, The Sandman by Neil Gaiman


https://casm-lang.org


github.com/casm-lang


twitter.com/casm_lang

This PhD thesis is entirely written in GNU Emacs *org-mode*, generated to LaTeX via GNU `emacs`, and type-set into this PDF document via `pdflatex`. The build process is based on GNU `make`. The statistical software `R` is used to generated LaTeX *TikZ*-based figures. All technical research artifacts of this thesis are contributed to the Corinthian Abstract State Machine (CASM) open-source project which is implemented in C++ and was initiated by the author on April 1st, 2014.

Declaration

Erklärung zur Verfassung der Arbeit

I hereby declare that this PhD thesis has been completed by myself by using the listed references only. Any sections, including tables, figures, etc. that refer to the thoughts, listed parts of the Internet or works of others are marked by indicating the sources.

Hiermit erkläre ich, dass ich diese Dissertation selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Location, Date)

(Philipp Paulweber)

Acknowledgments
Danksagungen

First, I would like to thank my doctoral supervisor Uwe Zdun who was guiding me through this thesis with so much support and always had an open ear for my ongoing problems and research challenges. He gave me no restrictions on how and in which direction I have developed this thesis over the last 6 years, and I am extremely thankful for that as well. Besides educating me in the research and academic procedures, I was more than honored to teach several lectures at the side of Uwe Zdun and support him with the practical parts of his courses as a university assistant, which increased my teaching skills significantly. Thank you for being my mentor, Uwe!

Second, I would like to thank Roland Lezuo and Andreas Krall for introducing me to the world of Abstract State Machines (ASMs) in 2012, when I worked on my master thesis at the Vienna University of Technology (TU Wien) back then. Since understanding the potential and mightiness of the ASM method, I was hooked on enriching the ASM-based language and tooling landscape by researching in the field of ASMs. Thanks, Roland and Andi for your friendship and all the great scientific discussions!

Next, I would like to express a really big thank you to Emmanuel Pescosta for being a long-term friend by now. He is part of the CASM open-source project and he supported my research with countless discussions and fun coding sessions especially for topics like the AST-based interpretation, the CASM IR, the concolic execution, or the object-oriented language constructs. Thanks Emme and see you soon in the upcoming CASM coding session!

Another big thanks goes to my former co-worker Konstantinos Plakidas for his friendship and his continuous motivating spirit during all the teaching ups and downs we had. Thanks Kostas, my friend and dive buddy!

Thanks to another co-worker, Christoph Czepa, for his friendship and all the great discussions we had about the statistical procedures regarding conduction controlled experiments. I want to include at this point Simon Tragatschnig, Jonannes Ferner, and especially Patrick Gaubatz. Thanks Kostas, Christoph, Simon, Johannes, and Patrick for being amazing friends!

A really big thank you to Sylvia Ennsberger for her countless hours of proof-reading my papers and this thesis. Thanks Sylvie!

Furthermore, I want to express my thanks to my former BSc thesis students Florian Hahn, Christian Lascsak, and Ioan-Valer Molnar who worked on smaller

CASM-related projects as well as my former MSc thesis student Jabob Moosbrugger for his implementation effort on the realization of the new TPTP support and concolic execution in CASM.

I would like to thank all students who participated in the two paper-and-pencil controlled experiments of the Distributed Software Engineering (DSE) and Advanced Software Engineering (ASE) courses in the summer term 2018 as well as the students of the Software Engineering 2 (SE2) course in the winter term 2018/19. Additionally, a thank you to the students and co-workers who participated in the eye-tracking experiment in the winter term 2018/19.

Next, I would like to thank the whole Rösch family for supporting me during a really long phase in this PhD thesis. Thank you Magdalena, Michael, Anna, Simon, and Aaron.

Moreover, I want to say a big thanks to all my friends as well who motivated me along the way as well with scientific and non-scientific stuff: Christoph Machreich, Jürgen Maier, Jakob Zwirchmayr, Stefan Mödlhammer, Dominik Köhle, Lukas Raneburger, Sandra Kirchner, Martin Fuchs, Jessica Conrnils, Simon Schweiger, Johanna Frei, Christoph Novak, Victoria Schuller, and Patrick Gruska!

I want to express my really big thanks to my Greek friends and diving buddies from Lepia in Rhodes, especially Antonis Karagiannis, Sakis Koniaris, and Manolis Iliochari for their amazing surface and sub-surface friendship. Thanks Toni, Sakis, and Manolis!

Thanks to all of my hockey club team members and friends from the WEV, especially Michael Vorlaufer, Clemens Takats, Stefan Nador, and Mario Kubeczka who helped me to clear my head by playing with me some great ice-hockey games in the Vienna Hockey League and 3rd Austrian Hockey League in the last 10 years.

Next, I would like to express a gigantic thank you to my parents Adele and Peter Paulweber as well as to my sisters Verena and Manuela Paulweber for always supporting me during this challenging time and having so much belief in me to achieve this step. Here I want to include Christian Wipplinger and the newest addition to the Paulweber crew, Maren Anna Paulweber. Thanks Mum, Dad, Manu, Very, Christian, and Maren for your support!

Finally, I would like to thank my girlfriend Nour Jnoub for her constant support, encouragement, and finding always the right motivating words and actions so that I could finish this thesis. Thank you, Nouri, I could not have done it without you!

Zunächst möchte ich mich bei meinem Doktorvater Uwe Zdun bedanken, der mich mit viel Unterstützung durch diese Arbeit geführt hat und immer ein offenes Ohr für meine aktuellen Probleme und Forschungsherausforderungen hatte. Er hat mir keine Grenzen gesetzt, wie und in welche Richtung ich diese Arbeit in den letzten 6 Jahren entwickelt habe, und auch dafür bin ich ihm sehr dankbar. Neben der Einarbeitung in die Forschungs- und Studienabläufe war es mir eine große Ehre, an der Seite von Uwe Zdun mehrere Vorlesungen zu halten und ihn bei den praktischen Teilen seiner Lehrveranstaltungen als Hochschulassistent zu unterstützen, was meine Lehrfähigkeiten erheblich gesteigert hat. Danke, dass du mein Mentor warst, Uwe!

Zweitens möchte ich Roland Lezuo und Andreas Krall dafür danken, dass sie mich 2012 in die Welt der abstrakten Zustandsmaschinen (ASMs) eingeführt haben, als ich damals an der TU Wien meine Masterarbeit schrieb. Seitdem ich das Potenzial und die Mächtigkeit der ASM-Methode erkannt habe, war ich Feuer und Flamme dafür, die ASM-basierte Sprach- und Tool-Landschaft durch Forschung im Bereich der ASMs zu bereichern. Danke, Roland und Andi, für eure Freundschaft und all die tollen wissenschaftlichen Diskussionen!

Als Nächstes möchte ich Emmanuel Pescosta ein großes Dankeschön dafür aussprechen, dass er mittlerweile ein langjähriger Freund ist. Er ist Teil des CASM-Open-Source-Projekts und hat meine Forschung mit zahllosen Diskussionen und unterhaltsamen Coding-Sessions unterstützt, insbesondere bei Themen wie der AST-basierten Interpretation, der CASM IR, der konkolischen Ausführung oder den objektorientierten Sprachkonstrukten. Danke Emme und bis bald bei der kommenden CASM-Codierungssitzung!

Ein weiteres großes Dankeschön geht an meinen ehemaligen Kollegen Konstantinos Plakidas für seine Freundschaft und seine ständige Motivation während all der Höhen und Tiefen, die wir in der Lehre hatten. Danke Kostas, mein Freund und Tauchkumpel!

Einem weiteren Mitarbeiter, Christoph Czepa, danke ich für seine Freundschaft und all die tollen Diskussionen, die wir über die statistischen Verfahren zur Durchführung kontrollierter Experimente geführt haben. Ich möchte an dieser Stelle Simon Tragatschnig, Jonannes Ferner und vor allem Patrick Gaubatz erwähnen. Danke Kostas, Christoph, Simon, Johannes und Patrick, dass ihr so tolle Freunde seid!

Ein wirklich großes Dankeschön an Sylvia Ennsberger für ihre unzähligen Stunden des Korrekturlesens meiner Papiere und dieser Arbeit. Danke Sylvie!

Darüber hinaus möchte ich mich bei meinen ehemaligen BSc Absolventen Florian Hahn, Christian Lascsak und Ioan-Valer Molnar bedanken, die an kleineren CASM-bezogenen Projekten gearbeitet haben, sowie bei meinem ehemaligen Diplomanden Jabob Moosbrugger für seine Implementierung und Realisierung der neuen TPTP Unterstützung und der symbolischen Ausführung in CASM.

Ich möchte mich bei allen Studierenden bedanken, die an den beiden Experimenten der Kurse DSE und ASE im Sommersemester 2018 sowie bei den Studierenden des Kurses SE2 im Wintersemester 2018/19 teilgenommen haben. Außerdem ein

Dankeschön an die Studierenden und Mitarbeiter, die an der Eye-Tracking Studie im Wintersemester 2018/19 teilgenommen haben.

Als nächstes möchte ich mich bei der gesamten Familie Rösch bedanken, die mich in einer wirklich langen Phase dieser Doktorarbeit unterstützt hat. Vielen Dank an Magdalena, Michael, Anna, Simon, und Aaron.

Darüber hinaus möchte ich mich auch bei all meinen Freunden bedanken, die mich auf meinem Weg mit wissenschaftlichen und nicht-wissenschaftlichen Dingen motiviert haben: Christoph Machreich, Jürgen Maier, Jakob Zwirchmayr, Stefan Mödlhammer, Dominik Köhle, Lukas Raneburger, Sandra Kirchner, Martin Fuchs, Jessica Cornils, Simon Schweiger, Johanna Frei, Christoph Novak, Victoria Schuller, und Patrick Gruska!

Ich möchte meinen griechischen Freunden und Tauchkumpels von Lepia auf Rhodos danken, insbesondere Antonis Karagiannis, Sakis Koniaris und Manolis Iliochari, für ihre erstaunliche Freundschaft an der Oberfläche und unter der Wasseroberfläche. Danke Toni, Sakis, und Manolis!

Danke an alle meine Eishockeyvereinsmitglieder und Freunde vom WEV, insbesondere Michael Vorlaufer, Clemens Takats, Stefan Nador und Mario Kubeczka, die mir geholfen haben, den Kopf frei zu bekommen, indem sie mit mir in den letzten 10 Jahren einige tolle Eishockeyspiele in der Wiener Eishockeyliga und der 3. österreichischen Eishockeyliga gespielt haben.

Als nächstes möchte ich ein riesiges Dankeschön an meine Eltern Adele und Peter Paulweber sowie an meine Schwestern Verena und Manuela Paulweber aussprechen, die mich in dieser herausfordernden Zeit immer unterstützt und so sehr an mich geglaubt haben, dass ich diesen Schritt schaffen kann. An dieser Stelle möchte ich auch Christian Wipplinger und das neueste Mitglied der Paulweber-Crew, Maren Anna Paulweber, erwähnen. Danke Mama, Papa, Manu, Very, Christian und Maren für eure Unterstützung!

Schließlich möchte ich meiner Freundin Nour Jnoub für ihre ständige Unterstützung und Ermutigung danken und dafür, dass sie immer die richtigen motivierenden Worte und Taten fand, damit ich diese Arbeit fertigstellen konnte. Danke, Nouri, ohne dich hätte ich es nicht geschafft!

Abstract
Kurzfassung

The demand for applying rigorous methods in the field of software engineering, hardware engineering, and systems engineering is still as high as ever. Over the last decades, several Domain-Specific Languages (DSLs) and tools were created to tackle different kinds of design challenges, capturing of requirements, and providing proper development support through code generation techniques for software as well as hardware fields. Despite the intensive investigation and domain exploration in their respective fields, most of the engineering methods lack proper techniques to reason about the specified design. This is where formal methods come into the picture of all engineering fields.

Industrial hardware engineering and development is mostly done or supported by formal method-based techniques, whereas in software and system engineering in general formal methods are still seen as a more academic discipline despite the effort already achieved by researchers all other the world. Especially for the software domain it becomes more and more relevant to industry because incorrect behavior, safety flaws, security breaches etc. can impact a company's reputation and market share or it can even cost human lives in safety critical or embedded applications. As a result, Model-Driven System Engineering (MDSE) methods were created which provide rich Model-Driven Development (MDD) techniques for a specific engineering domain.

The MDD technique includes a DSL and tool which consists of a dedicated parser, language validation analyzer, and code generator. Unfortunately, almost all MDSE methods are tailor-made formalisms to deal with a specific problem domain. Some of these techniques define and describe the language semantics through the code generation process, others allow for full or partially simulated systems' properties of the specified design before the creation of the actual system. Furthermore, a described design in one DSL cannot be reused or easily moved through rewriting into another DSL because of the completely different domain-specific abstraction level of the specification languages.

The Abstract State Machine (ASM) method is a state-based formal method which can be seen exactly as the missing piece in the described puzzle or specification dilemma above. It supports a domain independent way to capture a system's behavior regardless of whether the specified system is a software, hardware, or even a mixed software/hardware system. Based on an ASM, specified system reasoning can be applied in order to check certain system properties or even proof certain aspects of the

described system like safety constraints. Notable to mention is that ASM specifications are by default executable specifications, which means that any ASM-specified system can be simulated without even creating or deriving a concrete implementation. By definition, the latter can be derived either by refinement techniques or if supported through code generation just like MDSE/MDD does. Nevertheless, the tooling support for ASM-based specification languages regarding the interpretation (simulation) and compilation (code generation) lack modern compiler techniques and state-of-the-art language engineering.

A decade ago, a research project started to address both issues – interpretation and compilation – and was made public by the author through an open-source reimplementation named Corinthian Abstract State Machine (CASM). This PhD thesis describes the incrementally derived ASM-based compiler foundation and framework for CASM in order to research and explore further (optimizing) compiler and interpreter potentials as well as give the ability to research new language design concepts. In the course of this work, a novel ASM-based model-based transformation framework was elaborated by using a multi-level Intermediate Representation (IR) compiler design in order to achieve flexible software and/or hardware code generation with optimization focus in the foreground as well as fast execution (interpretation/simulation) as a second major research target. The defined ASM-based IR allows exploring and defining ASM-related compiler optimizations through a well-defined model and to provide a unified interface for other ASM-based language engineers and tool developers to e.g. reuse the implementation in CASM. Furthermore, an improved symbolic execution effort was achieved in this thesis for CASM as well, which is based in the ASM-based IR.

This PhD thesis reports on the investigation and introduction of an object-oriented language construct for ASM-based languages, which was derived in an incremental process applying two controlled experiments and one eye-tracking study. The first controlled experiment compared the understandability of three object-oriented abstractions introduced in an ASM-based language namely interfaces, mixins, and traits. Results showed that interfaces and traits have a similar good understanding, which lead to a follow-up controlled experiment investigating the usability of the two object-oriented language constructs interfaces and traits in the context of an ASM-based language syntax extension. A significant difference was discovered in the results, namely that the traits language construct is more usable compared to the interfaces language construct. Based on this insight, we conducted another controlled experiment in the form of an eye-tracking experiment for the trait-based language construct to obtain knowledge about the comprehensability of the syntax extension by analyzing the eye-gaze patterns as well as the eye fixation behavior. The outcome of these conducted studies is manifested in a novel trait-based object-oriented language construct in CASM, which provides an ability for ASM-based languages to easily describe ASM language properties in the language itself.

Die Nachfrage nach der Anwendung formaler Methoden im Bereich der Software-, Hardware- und System-Entwicklung ist nach wie vor ungebrochen. In den letzten Jahrzehnten wurden mehrere domänenspezifische Sprachen (DSLs) und Werkzeuge entwickelt, um verschiedene Arten von Entwurfsherausforderungen zu bewältigen, Anforderungen zu erfassen und geeignete Entwicklungsunterstützung durch Codegenerierungstechniken für Software- und Hardwarebereiche anzubieten. Trotz der intensiven Untersuchung und Erforschung der jeweiligen Domäne fehlt es den meisten Entwicklungsmethoden an geeigneten Techniken, um über das spezifizierte Design Aussagen zu treffen. An dieser Stelle kommen formale Methoden ins Spiel, die in allen verschiedenen Entwicklungs-Bereichen eine Rolle spielt.

Die industrielle Hardware-Entwicklung wird zumeist mit Hilfe formaler Methoden durchgeführt oder unterstützt, während formale Methoden in der Software- und Systementwicklung im Allgemeinen immer noch als eine eher akademische Disziplin angesehen werden, obwohl Forscher in aller Welt bereits große Anstrengungen unternommen haben. Insbesondere im Softwarebereich werden sie für die Industrie immer relevanter, da fehlerhaftes Verhalten, Sicherheitsmängel, Sicherheitsverletzungen usw. den Ruf und den Marktanteil eines Unternehmens beeinträchtigen oder bei sicherheitskritischen oder eingebetteten Anwendungen sogar Menschenleben kosten können. Infolgedessen wurden Methoden des modellgetriebene Systementwicklung (MDSE) entwickelt, die umfangreiche Techniken der modellbasierten Entwicklung (MDD) für einen bestimmten Entwicklungsbereich bieten.

Die MDD-Technik umfasst eine DSL und ein Werkzeug, das aus einem speziellen Parser, einem Sprachvalidierungsanalysator und einem Codegenerator besteht. Leider sind fast alle MDSE-Methoden maßgeschneiderte Formalismen für eine bestimmte Problemdomäne. Einige dieser Techniken definieren und beschreiben die Sprachsemantik durch den Codegenerierungsprozess, andere ermöglichen die vollständige oder teilweise Simulation der Systemeigenschaften des spezifizierten Entwurfs vor der Erstellung des eigentlichen Systems. Darüber hinaus kann ein in einer DSL beschriebener Entwurf nicht wiederverwendet oder durch Umschreiben in eine andere DSL übertragen werden, da die Spezifikationssprachen auf einem völlig anderen domänenspezifischen Abstraktionsniveau sich befinden.

Die abstrakte Zustandsmaschinen (ASM) Methode ist eine zustandsbasierte formale Methode, die genau als das fehlende Teil in dem oben beschriebenen Spezifikationsdilemma angesehen werden kann. Sie bietet eine domänenunabhängige Möglichkeit, das Verhalten eines Systems zu erfassen, unabhängig davon, ob das spezifizierte System ein Software-, Hardware- oder sogar ein gemischtes Software-/Hardware-System ist. Basierend auf einer ASM Spezifikation können Aussagen getroffen werden, um bestimmte Systemeigenschaften zu überprüfen oder sogar bestimmte Aspekte des beschriebenen Systems wie Sicherheitseinschränkungen zu beweisen. Erwähnenswert ist, dass ASM Spezifikationen standardmäßig ausführbare Spezifikationen sind, was bedeutet, dass jedes ASM spezifizierte System simuliert werden kann, ohne dass eine konkrete

Implementierung erstellt oder abgeleitet werden muss. Letztere kann per Definition entweder durch Verfeinerungstechniken oder durch Codegenerierung abgeleitet werden, wie dies bei MDSE/MDD der Fall ist. Dennoch mangelt es der Tool-Unterstützung für ASM-basierte Spezifikationssprachen in Bezug auf die Interpretation (Simulation) und Codegenerierung an modernen Compilertechniken und aktuellen Sprachkonzepten.

Vor einem Jahrzehnt begann ein Forschungsprojekt, das sich mit beiden Problemen - Interpretation und Codegenerierung - befasste und vom Autor durch eine Open-Source Reimplementierung namens Corinthian Abstract State Machine (CASM) veröffentlicht wurde. Diese Dissertation beschreibt die inkrementell abgeleitete ASM-basierte Compiler-Grundlage und das Framework für CASM, um weitere (optimierte) Compiler- und Interpreter-Potenziale zu erforschen und zu untersuchen sowie die Möglichkeit zu geben, neue Sprachdesignkonzepte zu erforschen. Im Rahmen dieser Arbeit wurde ein neuartiges ASM-basiertes modellbasiertes Transformationsframework unter Verwendung eines Compiler mit mehrstufigen Zwischendarstellungen (IR) erarbeitet, um eine flexible Software- und/oder Hardwarecodegenerierung mit Optimierungsfokus im Vordergrund sowie eine schnelle Ausführung (Interpretation/Simulation) als zweites großes Forschungsziel zu erreichen. Die definierte ASM-basierte IR erlaubt es, ASM-bezogene Compileroptimierungen durch ein klar definiertes Modell zu erforschen und zu definieren und eine einheitliche Schnittstelle für andere ASM-basierte Sprachen- und Werkzeugentwickler bereitzustellen, um z.B. die Implementierung in CASM wiederzuverwenden. Darüber hinaus wurde in dieser Arbeit auch eine verbesserte symbolische Ausführung für CASM entwicklet, die auf der ASM-basierten IR umgesetzt wurde.

Diese Dissertation berichtet über die Untersuchung und Einführung eines objektorientierten Sprachkonstrukts für ASM-basierte Sprachen, das in einem inkrementellen Prozess durch zwei kontrollierte Experimente und von einer Eye-Tracking Studie abgeleitet wurde. Das erste kontrollierte Experiment verglich die Verständlichkeit von drei objektorientierten Abstraktionen, die in einer ASM-basierten Sprache eingeführt wurden, nämlich Interfaces, Mixins und Traits. Die Ergebnisse zeigten, dass Interfaces und Traits ähnlich gut verstanden werden, was zu einem weiteren kontrollierten Experiment führte, in dem die Benutzerfreundlichkeit der beiden objektorientierten Sprachkonstrukte Interfaces und Traits im Kontext einer ASM-basierten Sprachsyntaxerweiterung untersucht wurde. Dabei wurde ein signifikanter Unterschied festgestellt, nämlich dass das Traits-Sprachkonstrukt im Vergleich zum Interfaces-Sprachkonstrukt besser nutzbar ist. Basierend auf dieser Erkenntnis führten wir ein weiteres kontrolliertes Experiment in Form eines Eye-Tracking Experiments für das Trait-basierte Sprachkonstrukt durch, um durch die Analyse der Blickbewegungsmuster sowie des Fixationsverhaltens der Augen Erkenntnisse über die Verständlichkeit der Syntaxerweiterung zu gewinnen. Das Ergebnis der durchgeführten Studien manifestiert sich in einem neuartigen Trait-basierten objektorientierten Sprachkonstrukt in CASM, das ASM-basierten Sprachen nun sogar die Möglichkeit bietet vorhandene und neue ASM Spracheigenschaften einfach in der ASM Sprache selbst zu beschreiben.

Contents

"We can only see a short distance ahead, but we can see plenty there that needs to be done." – Alan Turing

CHAPTER 1

Introduction

Nowadays, the spectrum of theories and tools to create various software and/or hardware systems in different design domain contexts is rather huge, starting from pure software development fields like web, desktop, server, kernel, and embedded-based all the way to pure hardware development fields like network, machine and chip-based. And in between are the joined hardware software co-design development fields, which require the knowledge, theories, tools and specification languages from both domains.

All of these design domain contexts have over time evolved different domain-specific and tailor-made practical solutions, sometimes some of them are even applicable in a reusable template mechanism. Additionally, to address topics like verification, validation, safety, fault-tolerance etc., all different research fields try to extend existing development techniques by checking partially specific properties and constraints. Especially in the field of embedded systems and Cyber-Physical Systems (CPS) with the need for safety-critical applications in the context of *Industry 4.0* [76], the design philosophy of the Internet of Things (IoT) [62], and *edge computing* [140], a strong need for a formal foundation is required to describe small to large scaling systems. Such a formal foundation shall consist of a precise specification language to capture the system's behavior, a description of environmental influences, and the possibility to use the system's description to perform various validation and verification checks.

There are several modeling techniques to serve as valid representatives to address and express a system's structure and behavioral description such as the Unified Modeling Language (UML) [55] standard[1] or the Systems Modeling Language (SysML) [56] standard[2]. UML was successfully used in different domains to describe (software) systems. A popular software architectural view where UML is used is the *4+1 View Model* by Kruchten [84]. SysML is based on the UML standard and provides an extension to express requirements and parametric properties [56] and can be seen as an industry de facto standard for Model-Driven Systems Engineering (MDSE).

[1] See `https://www.omg.org/spec/UML/About-UML` for the UML specification website.
[2] See `https://sysml.org` for the SysML open-source project website.

Depending on the target domain of the specified systems for UML or SysML, an implementation of the system has to be manually derived.

If a specification is defined in UML or SysML, it is specified conflict free (no syntactical violation), and a system structure of a possible implementation can be very easily derived through automated code generation. One famous example is the Eclipse Modeling Framework (EMF) [147] application[3], which provides a class diagram-based editor to compose a software system's data model. In this concrete example, the behavior of the system is not captured by the specification and has to be provided by hand-written code. Examples for MDSE frameworks for the embedded software domain which include specifications of systems' behavior are the SCADE [16] tool[4] or MATLAB/Simulink [107] suite[5]. There are even efforts to combine UML, SysML, and MATLAB to form an MDSE framework [153].

On the hardware side, the industry has established good development approaches along with Verification and Validation (VV) methods, which are built around Hardware Description Language (HDL) specifications like Very High Speed Integrated Circuit Hardware Description Language (VHDL) [96], Verilog [7], and SystemVerilog [150]. Independent from the various HDL versions, a designer in this hardware field has to obtain a certain level of knowledge (skills) to produce valid hardware designs which can be used for syntheses and programmed into a Field Programmable Gate Array (FPGA) or manufactured into an Application-Specific Integrated Circuit (ASIC). The problem with the existing HDLs is that they include additional language elements in order to create simulation environments along with the structural and behavioral descriptions of a system. The lack of abstraction and expressiveness of classical HDLs like VHDL and Verilog has been addressed in the last years by introducing new HDLs – Chisel [9] and SpinalHDL[6]. There has been some academic effort to provide a Model-Driven Development (MDD) approach by translating UML to HDL [37]. Nevertheless, the existing MDD approaches already have a specific target domain or environment in mind. So in order to provide a more generic MDD method and approach to address specification issues in the problem space ranging from high-level software to low-level hardware in mixed and non-mixed form, more abstract models have to be considered instead of models developed for a certain engineering purpose like SysML or SCADE.

A popular representative technique in academia and industry is to use state-based formal methods together with well-formed specifications. Prominent examples are Alloy [74], Event-B [1], Temporal Logic of Actions (TLA) [86], the Vienna Development Method (VDM) [19], and Z [75], which have been used in several academic and industrial scenarios as MDSE technique and for VV purposes. Nevertheless, these state-based formal methods are extremely mathematical and they require experts to specify the (software) system. If state-based formal methods are a good choice for an

[3]See `https://www.eclipse.org/modeling/emf` for EMF project website.
[4]See `https://www.ansys.com/products/embedded-software/ansys-scade-suite` for website.
[5]See `https://www.mathworks.com/products/matlab` for MATLAB project website.
[6]See `https://spinalhdl.github.io/SpinalDoc-RTD` for SpinalHDL project website.

MDD approach, what is a suitable specification language representation (notation) and abstraction level for novice, moderate, and professional *language users* [83] which is applicable to software, hardware, and systems design? The answer is a rigorous state-based method named Abstract State Machine (ASM) [63] [26].

The ASM theory and its formal methods provide the foundation to specify, analyze, and execute software and/or hardware systems. The main concepts are: (1) an executable *ASM specification* language which looks similar to pseudo code to express rule-based computations over algebraic functions with arbitrary data structures and type domains; (2) a *ground model* serving as a rigorous form of blueprint and reference model; (3) a stepwise *refinement* of the reference model by instantiating more and more concrete models which uphold the properties of the reference model [26]. Due to its mathematical foundation, ASM specifications can be analyzed using numerous existing rigorous verification and validation methods [25]. Based on the ASM language model by Gurevich [65], several tools with Domain Specific Language (DSL) implementations were created to solve application-specific problems [22]. There is a large diversity in the state-of-the-art of these applications[7], ranging from formal specification semantics of programming languages, such as those for Java [146] or VHDL [133], compiler back-end verification [90], software run-time verification [11], or software and hardware architecture modeling, e.g. of Universal Plug and Play (UPnP) [60] or even Reduced Instruction Set Computing (RISC) designs [72].

1.1 Motivation

So far, the state-of-the-art in ASMs does not yet support MDD structures and workflows well (see Section 1.7). Especially the functionality to execute ASM specifications symbolically is not covered in any other existing major ASM implementation like CoreASM [50] or AsmL [65]. In addition, the existing state-of-the-art does not address ASM-based specification reuse and retargeting to various execution environments and target languages. Despite its huge potential of existing ASM modeling languages, tools, and the well-defined ASM method, it did not achieve wide popularity. The work described in this thesis tries to increase the level of awareness for ASM-based modeling techniques in software engineering as well as hardware engineering communities.

In a previous research project called *Correct Compilers for Correct Application Specific Processors*[8] carried out by the Vienna University of Technology (TU Wien), a prototype ASM-based language named CASM[9] was created by Lezuo et al. [93] [91] to describe the Instruction Set Architecture (ISA) of RISC-based computer architectures. The further exploration of this ASM-based language towards optimized compilation and code generation to C/C++ by Lezuo et al. [94] had shown a huge potential and

[7] See `https://abz-conf.org/method/asm` for a list of various ASM applications.

[8] This research project was partially supported by the Austrian Research Promotion Agency (FFG) under contract 827485 and Catena DSP GmbH.

[9] The "C" was not named in the CASM acronym before 2014 and the name CASM was inspired by CoreASM because the roots of the CASM language were based in part on the CoreASM syntax.

research field for ASM-based compiler optimization, interpretation, and compilation (code generation) techniques. Unfortunately, the research project was discontinued and there was no plan to make the technical research artifacts publicly available to continue ASM-based research. In 2014, the author of this thesis started from scratch and provided a reimplementation of the CASM prototype as an open-source project and named the acronym for the first time as Corinthian Abstract State Machine (CASM)[10].

Lezuo [90] raised several open topics and research questions in the future work chapter in his dissertation, which were the starting point of this thesis which lead to the continuous improvement of the CASM language, interpreter, and compiler. One of the open core ideas was to apply the *"Translation Validation"* [90] approach by Lezuo on the CASM compiler itself because the prototype implementation *"[. . .] performs the optimizations directly on the [. . .]"* [90] Abstract Syntax Tree (AST). But this realization is easier said than done because this requires a concise ASM representation and an internal model of all supported ASM constructs inside the compiler framework (see Section 1.3, RQ_2). This implies that first, a proper definition and transformation for ASM specifications has to be explored in the form of an MDD approach for modern compiler design in order to address the *"Translation Validation"* [90] possibilities in CASM and for ASM languages in general (see Section 1.3, RQ_1).

Besides the above-mentioned compiler engineering related issues, Lezuo mentions another very interesting topic by stating that a *"CASM Object Model"* would be beneficial because the *"[. . .] CASM language currently operates on one global state. To create composable models, it would be advantageous to have objects. Rules would then operate on their object's state. An open research question however is the composition of objects, especially considering the transactional semantics of ASM.'* [90] This stated research question was one of the main motivations for the different language construct investigations in this thesis because to address this properly, several aspects have to be elaborated. First of all, the question which kind of object-oriented concept is the appropriate one for ASM-based languages had to be answered. As a next step, how is it possible to describe object-oriented ASM specifications without compromising the ASM theory, which raises the need for a formalization or refinement of object-oriented ASMs towards the basic ASM definition from Börger [26].

1.2 Research Methods

For the technical part, this PhD thesis uses the principles of the *Design Science Research* method [69] [155], as a research approach, which is also known as the *Constructive Research* method [124] for the practical tasks and technical contributions. The *Design Science Research* method defines the following steps: (1) a research question is posed, which can include simplified assumptions; (2) a development and/or evaluation cycle is continuously repeated until a satisfactory solution for the research

[10] See `https://casm-lang.org` for open-source project website.

question has been obtained; and (3) altering the research question either by removing assumptions or by tightening the research question itself.

Besides the *Design Science Research* method, controlled experiments are used for the empirical part of this PhD thesis and to solve the related research problems listed in Section 1.4 and derive the answers to the stated research questions in Section 1.3. Two different kinds of controlled experiments are used to obtain empirical evidence.

First, we performed two controlled experiments in which participants process a printed survey. These controlled experiments are designed, executed, and described in this PhD thesis according to the guidelines by Kitchenham et al. [82] [81], Wieringa [156] and Wohlin et al. [158], as well as the specific software architecture research and empirical research guidelines by Falessi et al. [49] on how to conduct such studies in order to derive empirical evidence and by applying proper statistical analysis procedures on the obtained quantitative data.

Second, we used eye-tracking experiments [110] for investigating human interaction with the given stimuli by measuring eye movement. Based on the latter, so called eye-gaze patterns can be derived by analyzing the eye fixations, which leads to new empirical evidence.

1.3 Research Questions

Based on the motivation described in Section 1.1, this PhD thesis wants to explore modern compiler engineering methods for an ASM-based specification language using MDD techniques in the form of multi-level Intermediate Representation (IR)s. Moreover, besides the compiler engineering aspects, a major concern in this thesis is the exploration and introduction of an object-oriented abstraction into the ASM language implementation CASM. We state the following research questions (RQ_n):

RQ$_1$ How to translate ASM specifications capturing the structural and behavioral properties of described systems into software, hardware, and/or mixed software and hardware applications and artifacts using an MDD approach?

RQ$_2$ How can the optimization potential regarding execution of ASM specifications as well as the translation validation capabilities in an ASM-based compiler be addressed in a unified manner?

RQ$_3$ How well are object-oriented abstractions understandable, applicable, and comprehensible by novice, moderate, and expert engineers by using a corresponding language construct syntax extension in an ASM-based specification language?

1.4 Research Problems

In order to be able to derive an MDD-based transformation of ASM specification in a modern compiler, we have to deal with several design and implementation concerns either for the *language user* [83] or *language engineer* [83] perspective (see problem space elaboration in Section 1). The following research problems (P_n) are addressed in this PhD thesis:

$\mathbf{P_1}$ *Lack of Statically Strong Typed ASM Specification Languages*

This problem relates to RQ_1 because in order to cover all engineering domains a precisely typed ASM language is needed to fulfill the needs for software and hardware engineering. Current ASM implementations only include mathematical abstract data structures. Due to the fact that every ASM-based language will eventually be executed by a real machine, a term, expression, or even a value will have a concrete type. Even Gurevich [64] suggested his ASM language definition lacks explicit typing, and it would be more practical to introduce such.

$\mathbf{P_2}$ *Lack of Abstractions for Reusability in ASM Specification Languages*

In order to answer RQ_1 and RQ_3, existing ASM languages have to be extended with a proper type abstraction (object-oriented concept) to enable structuring and reuse of ASM specifications, which is mentioned by Börger [24] in his latest article. Furthermore, to improve the (re)usability, a module and import system is missing in existing ASM languages.

$\mathbf{P_3}$ *Lack of Transformations to Different Execution Contexts and Environments*

The RQ_1 cannot be answered with existing ASM implementations because besides the simulation of ASM specifications, a generic code generation (compilation) to arbitrary target execution contexts and environments from high-level software, low-level software, high-level hardware, to low-level hardware of ASM specifications is missing.

$\mathbf{P_4}$ *Lack of Performance-related Optimizations for the Execution Contexts*

As pointed out by Lezuo et al. [94], there is a huge optimization potential (e.g. execution performance) for statically typed ASM-based specification languages in the form of static compiler analyses and transformations by eliminating redundancy. No ASM compiler infrastructure exists yet to provide rewriting and optimization techniques for ASM-based languages. This problem is part of RQ_2 because it affects the generated code and should be addressed in a unified manner.

$\mathbf{P_5}$ *Lack of Translation Validation Support inside of ASM Language Tooling*

In order to answer the remaining part of RQ_2, the existing tool landscape of ASM implementations does not provide any ability to perform translation validation capabilities inside an ASM language tooling in an automatic fashion.

P_6 *Lack of Empirical Evidence Understanding ASM Specification Languages*

According to the state-of-the-art, so far no study has investigated the reading and writing of ASM-based specifications as well as evaluated the understandability or usability of ASM language syntax, features, and/or constructs. This research problem relates to RQ_3.

1.5 Research Contributions

This section provides an overview of all contributions of this PhD thesis, which are linked to their corresponding research problems described in Section 1.4. All technical implementation artifacts are published and contributed to the CASM project[11].

In the course of this thesis, the contributions were published in formal method communities and conferences like the International Conference on Rigorous State-Based Methods (ABZ)[12] as well as in prominent journals like ElseVier Journal of Systems and Software (JSS)[13] or ACM Transactions on Software Engineering and Methodology (TOSEM)[14]. Besides this, one contribution was published in the eye-tracking community and workshop named Eye Movements in Programming (EMIP)[15] and another contribution was published in the hardware community and conference called Asynchronous Circuits and Systems (ASYNC) [16]. The contributions (C_n) published in the course of this PhD thesis are:

C_1 Reusable and Retargetable ASM Specifications (ABZ'16) [123]

This contribution addresses P_3 by introducing a new compiler infrastructure design for CASM, which uses a model-based transformation approach to separate the CASM front-end and different back-end implementations by multiple IR levels.

C_2 Asynchronous Logic Design Approach (ASYNC'19) [115]

This contribution is part of P_3 by envisioning the design and development potential of the proposed reusable and retargetable model-based transformation approach of C_1.

C_3 Precise Type System and IR for ASM Compilers (ABZ'18) [117]

To solve P_1 as well as in part P_3 and P_4, a specific CASM IR for ASM languages was designed and implemented to provide all ASM related properties in a uniform way decoupled from ASM language dialects to design and implement ASM-based compiler analysis and transformation (optimization) passes in an independent manner.

[11] See `https://github.com/casm-lang` for the open-source project and organization website.
[12] See `https://abz-conf.org` for ABZ conference and community website.
[13] See `https://www.journals.elsevier.com/journal-of-systems-and-software` for website.
[14] See `https://tosem.acm.org` for the journal website.
[15] See `http://www.emipws.org/workshop/emip-2019` for the workshop website.
[16] See `http://async2019.jp` for the conference website.

C_4 Symbolic Promotion and Concolic Execution for ASMs (ABZ'21) [118]

As a first step to solve P_5, a new concolic execution was implemented for CASM with a novel symbolic function promotion for ASM functions which uses a model-based Thousands of Problems for Theorem Provers (TPTP) [149] trace generation approach to create symbolic traces in a generic way. This contribution generalizes the prototype work of Lezuo [90].

C_5 Type Abstractions in ASM Specifications (ABZ'20) [116]

This contribution addresses P_2 and in part P_6 by providing a new type abstraction (object-oriented concept) to the CASM language to specify structure and behavioral elements in a reusable manner.

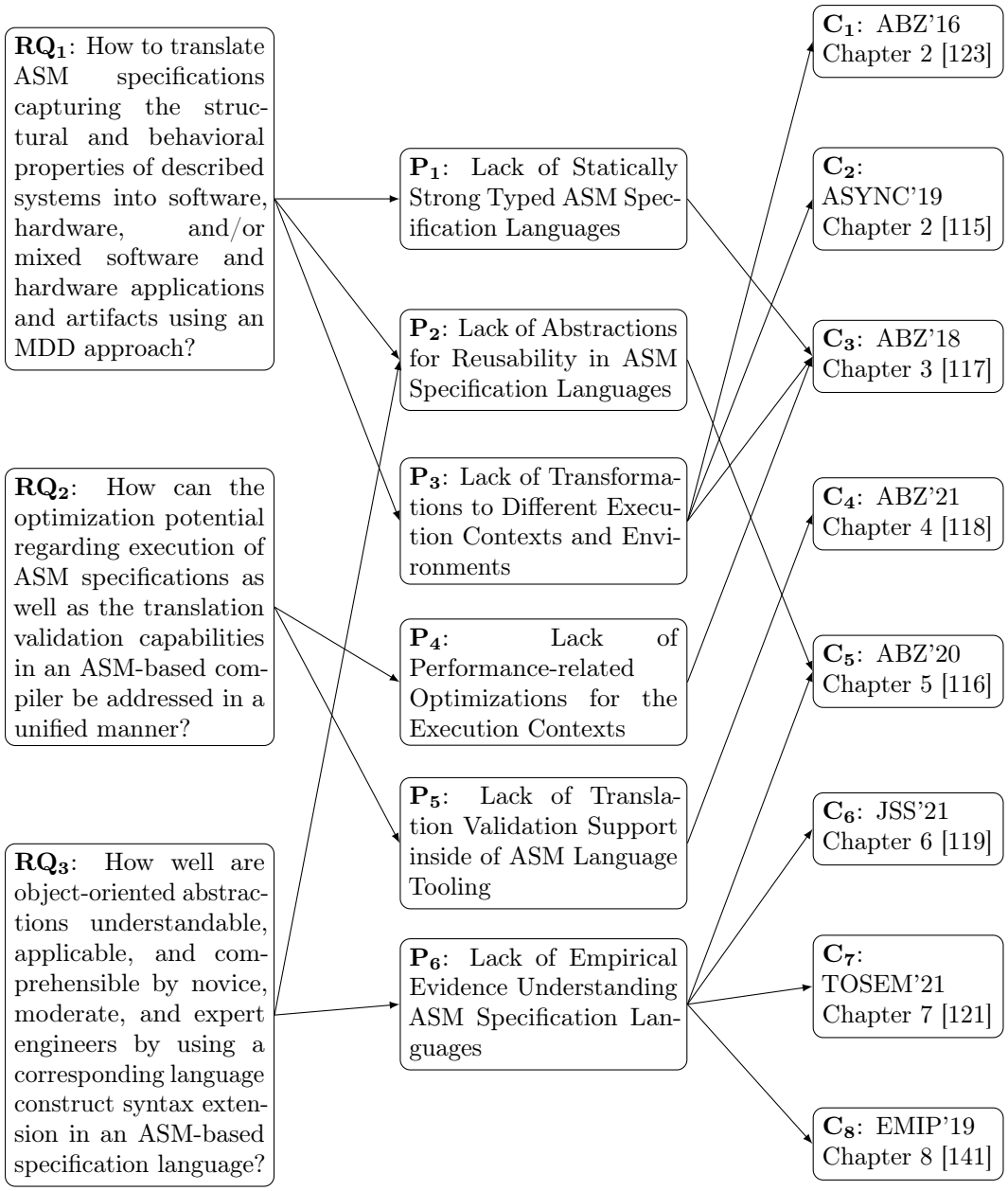


Figure 1.1: Research Overview

C$_6$ Understandability Study for Type Abstractions in ASMs (JSS'21) [119]

To solve in part P$_6$, this contribution investigates how well three different object-oriented ASM syntax extensions are understandable through a controlled experiment where participants needed to read (comprehend) ASM specifications and answer a survey about the structural and behavioral properties of the given stimuli.

C$_7$ Usability for Object-Oriented Concepts in ASMs (TOSEM'21) [121]

This contribution addresses in part P$_6$ by providing significant results in a direct comparison of two ASM-based type abstractions (object-oriented concepts) to the CASM language where participants received an informal specification of a system as textual description and they had to write (specify) the corresponding ASM specification using a given ASM-based object-oriented abstraction.

C$_8$ Eye-Tracking Study on Language Concept in ASMs (EMIP'19) [141]

To conclude and solve in part P$_6$, in this contribution an eye-tracking experiment was conducted to explore the eye-gaze behavior and fixation patterns of participants when they are comprehending an ASM-based specification language with a concrete object-oriented abstraction and language syntax extension in place.

Research Overview

Figure 1.1 provides a complete overview of the research carried out in this PhD thesis by visualizing the research questions (RQ_n), the research problems (P_n), and the contributions (C_n) with their connections and relationships between each other.

1.6 Abstract State Machines

In 1995 Gurevich [63] described the ASM theory, which is a well-known formal method based on transition *rules*, *agents*, and mathematical *function* states that can be used to specify arbitrary algorithms, applications or even whole systems. The mathematical *function* state is defined with a corresponding type relation. *Agents* are able to execute *rules*, which enables the producing of *updates* resulting in a change of the global mathematical *function* state. Since the appearance of the ASM theory, several definitions and implementations of ASM-based languages, interpreters, and compilers have been created. However, all of them focus mainly on the analysis of ASM specifications for certain properties or on software-sided simulations. The CASM language [91] and project[17] focuses on enabling a way to not only analyze and simulate ASM-based specifications. One of the main research objectives is to establish a generic transformation of CASM specifications to various target languages and execution

[17] See `https://casm-lang.org` for CASM project website.

environments including the software and hardware domain through a model-based transformation approach [123]. Important to note is that in contrast to already existing hardware-based languages or IRs (Chisel [9], Flexible Intermediate Representation for Register Transfer Level (FIRRTL) [73], or YoSys [159]), CASM does not include any assumptions of the resulting circuit style (synchronous design, clock etc.). Therefore, the same CASM input specification can be (re)used and (re)targeted. In order to achieve this goal, multiple compiler IRs [117] were introduced to separate the languages' own run-time implementation of a certain target environment or language. A possible asynchronous hardware target environment could be the *link and joint model* by Roncken et al. [131].

Basic and Turbo ASMs

The *Basic ASM* is defined by Börger and Stärk [26] as consisting of abstract states, signatures, locations, updates, update sets, firing of updates, terms, formulas, transition rules, ASM move, and ASM run. The abstract states are algebraic structures and can be seen as abstract memories. A location is a concrete position (argument vector of a function) in the abstract memory. A signature (vocabulary) consists of all function names in an ASM. Every function can be classified either as *derived*, *static*, or *dynamic*[18]. Updates are location value (content of a function at a certain location) pairs, which are collected in an update set during an ASM move. After an ASM move, the collected updates from the update set are applied (fired) to the (global) function states. An ASM run consists of one or multiple ASM moves. The transaction rules in a *Basic ASM* are: (1) *skip* to perform no operation; (2) *update* to create a new update $l := v$ of an n arity function location $l = (f, a_1, \ldots, a_n)$ and a value v; (3) *block* to express bounded parallelism; (4) *conditional* to express branching; (5) *let* to express variable bindings; (6) *forall* to express parallel execution over a universal quantifier; (7) *choose* to express indeterministic choice; and (8) *call* to invoke sub transaction

[18]The dynamic functions can be further classified into *in*, *controlled*, *shared*, and *out*.

```
1  Identifier          ::= /* symbol names */.
2  Type                ::= Identifier | /* other type representations */.
3  Literal             ::= /* literal representations */.
4  Term                ::= Literal | /* other terms and expressions */.
5  Rule                ::= BlockRule | CallRule | LetRule | ImportRule | /* other rules */.
6  BlockRule           ::= '{' Rule '}'.
7  UpdateRule          ::= Identifier [ '(' Term (',' Term)* ')' ] ':=' Term.
8  LetRule             ::= 'let' Identifier '=' Term 'in' Rule.
9  CallRule            ::= Identifier [ '(' Term (',' Term)* ')' ].
10 ImportRule          ::= 'let' Identifier '=' 'new' Type 'in' Rule.
11 DomainDefinition    ::= 'domain' Identifier.
12 FunctionDefinition  ::= 'function' Identifier ':' [ Type
13                         ( '*' Type )* ] '->' Type [ '=' Term ].
14 DerivedDefinition   ::= 'derived' Identifier '(' Identifier ':' Type
15                         ( ',' Identifier ':' Type )* ')' '->' Type '=' Term.
16 RuleDefinition      ::= 'rule' Identifier '(' Identifier ':' Type
17                         ( ',' Identifier ':' Type )* ')' [ '->' Type ] '=' Rule.
18 // ... other grammar rules of ASM language
```

Listing 1.1: Turbo ASM Syntax (Excerpt)

rules.

The *Basic ASM* gets extended to a *Turbo ASM* [26] by adding additional transaction rules: (1) *seq* to express sequential composition transaction rules; (2) *iterate* to express fix-point iterations; (3) named *rule* construct to create submachines; and (4) *local* rule to define local functions.

We use the following mathematical notation: (1) Σ defines a finite set of all function names and f denotes a function name of Σ ($f \in \Sigma$); (2) $\mathcal{D}$ defines a finite set (subset of Σ) of function names classified as derived and d denotes a derived function name of $\mathcal{D}$ ($d \in \mathcal{D} \ \wedge \ \mathcal{D} \subseteq \Sigma$); (3) $\mathcal{R}$ defines a finite set of rule names and R denotes a rule name of $\mathcal{R}$ ($R \in \mathcal{R}$); and (4) $\mathcal{U}$ defines a finite set of universe (domain) names and X denotes a type name of $\mathcal{U}$ ($X \in \mathcal{U}$).

We only present the necessary Turbo ASM language [26] elements in order to understand the given transformation for the trait-based construct in Chapter 5. The necessary syntax rules are defined in Listing 1.1.

The grammar rules `Identifier` and `Type` represent a symbol name in ASM, which can be a function name (e.g. $f, d \in \Sigma$), a named rule (e.g. $R \in \mathcal{R}$), a variable which appears in the `let` rule (e.g. v), or a type name of a given universe (domain) (e.g. $X \in \mathcal{U}$). `Literal` represents a constant of a certain universe (domain), e.g. the Boolean algebra $\Sigma_{Boolean}$ consists of the constants $undef$, $false$, and $true$. The grammar rule `Term` contains either a given constant literal or a formula.

`Rule` represents a *transaction rule* [26] of an ASM. `BlockRule` represents bounded parallelism of rules R and S ($\{R, S\}$). `UpdateRule` represents the creation of a new update u consisting of a location l and a value (term) t pair, which gets placed into the update set ($u = (l, t) \ \wedge \ l = (f, a_1, \ldots, a_n)$). `LetRule` represents a variable binding by creating an equation consisting of a variable name v and a term t and evaluating a rule R in the scope of the variable v ($let \ v = t \ in \ R$). `CallRule` represents invocation of a submachine rule R with call-by-value semantics for the parameters p. Note that by default, the *Basic ASM* uses call-by-name semantics. In this definition, we use call-by-value semantics, which evaluates the parameters in the current state of the ASM before the evaluation of the submachine rule. This can be modeled by binding all the parameters first through a *let* rule and evaluating the rule R afterwards ($R(p_1, \ldots, p_n) \Leftrightarrow let \ v_1 = p_1, \ldots, v_n = p_n \ in \ R(v_1, \ldots, v_n)$). `ImportRule` is a syntactic abbreviation for the *import* rule [26] to extend a domain X with a *"completely fresh symbol"* [26] x from the *reserve* of an ASM to increase the workspace of a specified algorithm and use the new symbol x in the evaluation of rule R ($let \ x = new \ X \ in \ R \Leftrightarrow import \ x \ do \ \{ \ X(x) := true, R \ \}$).

`DomainDefinition` defines a new unique universe (domain) name X, which gets added to $\mathcal{U}$ ($X \in \mathcal{U}$). `FunctionDefinition` defines a new unique function name f, `DerivedDefinition` defines a new unique derived function name d, and `RuleDefinition` defines a new unique rule name R, where all have a non-negative arity n of argument types $\{X_1, \ldots, X_n\}$ and a target type X_t ($f, d, R \in \Sigma \ \wedge \ n \in$

$\mathbb{N}_{\geq 0} \wedge \{X_1, \ldots, X_n\}, X_t \in \mathcal{U})$ [19]. Since the target type of rule definitions is optional in the grammar, we use a special domain named *Void* to type the target type in case of absence for all named rule definitions.

1.7 Related Work

This section covers the state-of-the-art by presenting the related work of ASM language implementations and ASM transformation and code generation approaches.

ASM Languages and Implementations

One of the best-known ASM implementations is the *Asmeta*[20] tool-set with the *AsmetaL* language [58]. The core of *Asmeta* is designed and implemented using the EMF *Ecore* meta-model[21]. Based on the *Ecore* meta-model, the ASM language model of *Asmeta* is directly described as an instance (model). Therefore, the execution and precise calculation of the implemented ASM simulator is bound to the run-time implementation of the *Ecore* meta-model and its EMFs Java interface realizations.

Another notable ASM language and implementation is *CoreASM*[22], originally developed by Farahbod et al. [50]. The focus of CoreASM is to provide a flexible and extensible ASM implementation and to be as near as possible to the described ASM method by Börger [26]. CoreASM is implemented in Java and its IR and run-time is directly bound to the Java Virtual Machine (JVM).

Microsoft research designed and implemented an ASM language named *AsmL*[23] [65]. AsmL is implemented and based to the `.NET` framework.

Arcaini et al. [5] proposed a Unified Abstract State Machine (UASM) language syntax. Their approach is to unify the front-end ASM syntax representation. Similar to the ASM language proposed by Anlauff [3], the eXtensible ASM (XASM) language[24] compiles XASM specifications to C.

Ouimet and Lundqvist [113] presented another ASM language named Timed Abstract State Machine (TASM). Their language and simulator approach focused on a real-time ASM notation.

Lezuo et al. [93] introduced the CASM language in 2013. The origin of this language was that all the (publicly available) existing ASM tools were impracticable for industrial sized applications [91]. The tool-chain presented by Lezuo et al. [94] [90] focuses, like the other ASM designs, only on the input specification itself, thus those research results were not directly usable by other ASM-based language frameworks.

[19] $\mathbb{N}_{\geq 0} = \{n \in \mathbb{N} \mid n \geq 0\}$
[20] See `https://asmeta.github.io` for Asmeta project website.
[21] See `http://eclipse.org/modeling/emf` for EMF project website.
[22] See `https://github.com/coreasm` for CoreASM project website.
[23] See `http://asml.codeplex.com` for AsmL project website.
[24] See `http://sourceforge.net/projects/xasm` for XASM documentation website.

ASM Transformation and Code Generation

Different representation and transformation approaches have been investigated in the *AsmGofer* language by Schmid [136], which is based on the programming language Gofer [78], and the *ASM Workbench* with the *ASM-SL* language introduced by Del Castillo [43], which is implemented in Standard ML [68]. The *ASM-SL* has been explored further by Schmid [135] to represent and encode specifications in C++. The translation (compilation) scheme was limited to a *double buffering* concept and therefore unable to encode mixing sequential and parallel rules.

Another transformation scheme for ASMs was presented by Bonfanti et al. [20] to represent and encode *AsmetaL* specifications in C++ code targeting Arduino platforms. Their code generator directly converts the ASM specification to the desired target language and run-time environment. By targeting a different target run-time environment, platform, or architecture, the encoded and implemented ASM behavior would have to be re-implemented in every code generator.

Notable to mention is the effort by Sinha et al. [142] to use CoreASM as IR for HDL synthesis of synchronous digital circuits. By only implementing a certain sub-set of the CoreASM language rules and providing additional specialized ASM rules to introduce clock-driven behaviors, the approach results in a very customized one and does not provide a generic solution as this work will address.

1.8 Structure of this Thesis

The remainder of this PhD thesis is structured as follows: Chapter 2 provides an overview of the model-based transformation approach used in CASM to design and implement a modern and state-of-the-art compiler framework in order to provide a foundation for researching ASM-based language constructs, language analysis and transformation techniques, interpretation, and code generation.

In Chapter 3 the CASM IR is introduced, which is part of the CASM compiler framework and allows capturing ASM specifications in a unified manner as well as provides the basis for ASM-aware front-end (syntax) independent compiler optimizations.

Chapter 4 describes the improved concolic execution along with an ASM symbolic function promotion analysis approach to improve the symbolic trace generation and outlines the corresponding implementation in CASM.

Chapter 5 introduces a novel ASM-based object-oriented abstraction and corresponding language construct by defining its syntax and semantics and describing the actual implementation in the CASM project. This object-oriented abstraction was the result of three conducted controlled experiments, which are described in the following Chapters 6, 7, and 8.

Chapter 6 describes the pilot study to gather insights about the understandability of three different object-oriented abstractions which could be introduced into an ASM-based language where CASM was used as an ASM representative.

Based on the outcome of the previous chapter, Chapter 7 describes the closer comparison of a controlled experiment between two object-oriented language constructs with the focus on their usability and applicability for ASM-based languages with CASM as an example language.

Chapter 8 describes an eye-tracing experiment, where the more applicable language construct of the previous study in Chapter 7 was investigated in detail by analyzing eye-gaze behavior, fixation points, and cognitive loads visible through certain timing behavior.

All chapters contain a small conclusion section. Therefore, Chapter 9 provides an overall summary of the contributions and concludes this PhD thesis.

"Hardware eventually fails. Software eventually works." – Michael Hartung

CHAPTER 2

Model-Based Transformation

As described in the previous chapter in Section 1.6, the ASM theory is a way to specify algorithms, applications and systems in a formal model. Recent ASM languages and tools address either the translation of ASM specifications to a specific target programming language or aim at the execution in a specific environment. In this chapter[1] we outline a model-based transformation approach supporting (1) the specification of applications or systems using the CASM modeling language and (2) retargeting those applications to different programming language and hardware target domains. An intermediate model is introduced, which not only captures software-based implementations, but also the generation of hardware-related code in the same model. This approach offers a new formal modeling perspective onto modular, reusable and retargetable software and hardware designs for the development of embedded systems. We provide a short overview of our CASM compiler design as well as the retargetable model-based approach to generate code for different target domains. Furthermore we discuss the possible accelerating of asynchronous logic in the hardware developing community through the mentioned model-based transformation approach.

2.1 Introduction

Since 1995 where Gurevich has described the ASM theory [63], many approaches have been proposed to interpret, execute, translate, verify and validate ASM specifications (summarized by Börger [23]). Generally speaking all available (public) tools either aim to integrate an ASM language into a specific (software) platform system/framework or focus on a domain specific purpose. We want to enlarge the scope of ASM language tools and provide a general purpose modeling system for the CASM modeling language (introduced by Lezuo et al. [91]). Such a system will enable us to specify arbitrary applications/systems in this language and translate them into one or multiple

[1] The content of this chapter is a revised version of the ABZ'16 paper [123] and an adapted version of the ASYNC'19 paper [115].

programming language and hardware target domains. To the best of our knowledge, such a generic translation does not yet exist.

Furthermore, not only is the focus of our investigation not limited to translations to several software environments, it also includes the idea to translate CASM specifications to different HDL contexts. This will enable us to even describe electronic circuit designs with CASM specifications and will result in a broad range of applications from specifying small embedded applications up to RISC microprocessors or even complete System-on-Chip (SoC) designs in a formal way.

The CASM modeling language was designed and used by Lezuo et al. [91] to describe the semantics of machine languages. Moreover, they performed compiler correctness proofs through the usage of the ASM machine models and compiled specifications written in this language into efficient C/C++ applications [94].

Unlike other ASM specification languages such as AsmL [65] or CoreASM [50], CASM currently consists of a small grammar and a static, strong type system, and it only supports a subset of rules from the CoreASM modeling language. The static, strong type system allows to optimize such specifications. Initially, the syntax of CASM followed CoreASM, but over time it diverged significantly (differences to other ASM modeling languages are described by Lezuo et al. [91]). Due to the (currently) small grammar, the optimization potential and simplicity, the CASM modeling language is a good fit for our effort to retarget ASM specification.

Before we go into details, let us review the design of the compiler infrastructure proposed by Lezuo et al. [94]. Figure 2.1 depicts the translation process. The parsed CASM specifications are transformed into an AST, and after that type checks and type annotations are performed. Several static optimizations are performed to eliminate run-time overheads. All transformations which need run-time specific calculations and knowledge are redundantly implemented in the AST-based optimizations. The compiler directly emits C/C++ code in the next step, which then gets compiled and linked against the C/C++ run-time library. Important to mention here is that the generated code and the run-time are not synchronized in their implementation state.

The design in Figure 2.1 is not a retargetable infrastructure. That is, in this design, the existing code emitter and run-time implementation need to be checked for correctness, and it must be tested that the execution and calculation of the generated C program equals the specified CASM input specification. If we would retarget this design to different software or hardware environments, we would have to check for the code emitter and run-time implementation again for each new environment that

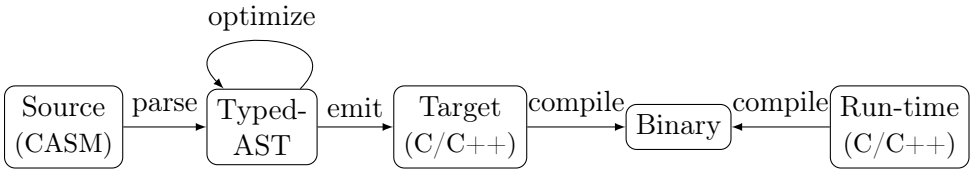


Figure 2.1: CASM Compiler with C/C++ Back-end

the calculation behavior of the generated target equals the specified CASM input specification.

The emitting stage depicted in Figure 2.1 is the main focus of our approach. Our solution to this *retargetable CASM specification problem* is to abstract the run-time and the emitted code in a specific calculation model. This will allow us to check the transformation from the CASM model to this specific computational model once. And for every new target environment (software or hardware) we add to the compiler, only the transformation has to be checked from the specific calculation model to the new target environment. Therefore, we can develop several different code emitter implementations hand-in-hand with *one* run-time implementation and *one* CASM transformation implementation.

This approach enables us to create and generate reusable and retargetable software or hardware artifacts. Those artifacts are self-contained because in our approach we even include the full CASM run-time in the generated artifacts. Hence, the generated artifacts of CASM input specifications can be deployed without further libraries or dependencies. The latter is very important when it comes to hardware-related generated code, because it will not only ease the integration in other hardware designs, but will also allow HDL compilers to fully optimize the generated HDL code on module level.

2.2 Retargetable Approach and Models

The design of our CASM implementation follows a strict model-based transformation approach to overcome the *retargetable CASM specification problem*. Figure 2.2 depicts our model-based transformation approach where we introduce two models – the IR and the Emitting Language (EL) model.

Intermediate Representation Model

The IR is a full CASM semantics aware model which will be described in detail in Chapter 3. It is to analyze and optimize the input specification. An instance of this model is created during the AST to IR transformation which is the first transformation

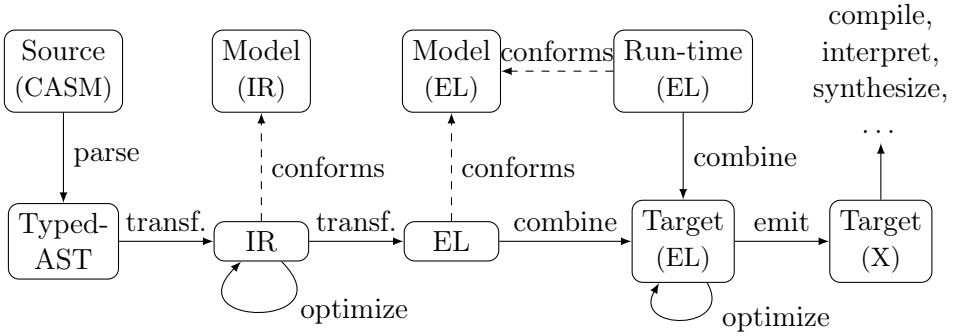


Figure 2.2: CASM Compiler with Model-Based Transformation

step depicted in Figure 2.2. The IR model consists of two important characteristics – parallel/sequential Control Flow Graph (CFG) (introduced by Lezuo et al. [94]) and explicitly modeled operations which are not covered in the AST representation from Lezuo et al. [94] e.g. the location of a ASM state function. The proposed ASM specific *lookup and update elimination* optimizations by Lezuo et al. [94] are planned to be implemented at this level. Software back-ends will profit from those optimizations to be able to execute the specifications much faster (as shown in [94]). Furthermore, we strongly believe the hardware back-ends will benefit from the proposed optimizations too. Because the generated HDL code will result in a less complex digital design by reducing the number of performed calculations just like it applies to the generated software code.

Emitting Language Model

An instance of this model is created during the IR to EL transformation of the IR instance which is depicted in Figure 2.2 as second transformation step. It allows us to express the CASM run-time and the CASM input specification in a CASM semantics unaware fashion. Thereby we are forced to find generic abstract language constructs for the EL model which allow us to express calculations, procedures and sequential and parallel execution behavior. Figure 2.3 depicts the class diagram of the EL model.

The EL model is designed to make the mapping to different software/hardware targets easier, but this generic abstraction does not come without limitations. For example the only data type allowed in the EL model is a bit-precise (arbitrary bit-size) integer value (*Bit-type*) to enable a clean translation to HDL data types. To represent complex or compound data a structure concept is available in the EL model as well to create records of several bit-precise integer values.

The overall model construct is a *Module* which can contain besides *Constants*, *Variables*, *CallableUnits* also explicitly defined *Memory* blocks. The *Memory* blocks are used to properly allocate the appropriate amount of wiring and memory storage in the generated HDL designs. The difference between a *Memory* and *Variable* storage is that *Variables* are translated to HDL designs as plain registers and only permit a single write access.

Memory blocks permit multiple write access. We assume in the EL model that

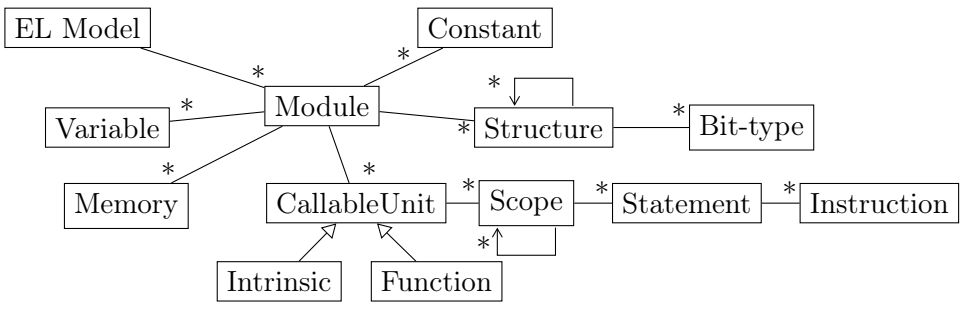


Figure 2.3: Emitting Language Model Class Diagram

each write access is mutually exclusive. The latter is important, because the model allows the construction of mixed parallel and sequential *Statement* blocks.

CallableUnits are divided into two procedural constructs – *Functions* and *Intrinsics*. Software back-end languages like C, Python etc. use this differentiation to emit efficient target language code, which can be used by the target compiler/interpreter to optimize the execution of the program. Hardware back-end languages can derive a differentiation between behavioral descriptions and computational logic blocks. At this point, another important EL model characteristic is that a *CallableUnit* does not have a “return" value. All incoming and outgoing data of a *CallableUnit* has to be explicitly defined through “in" and “out" parameters. Hence, software back-ends will use this to generate “call-by-reference" constructs and hardware back-ends generate direct component wiring. All *CallableUnits* can contain mixed parallel and sequential *Scopes* to define a concurrent and sequential calculation hierarchy. Every *Scope* in the EL model can contain several *Statements*.

A *Statement* can either be a “trivial", “branch" or “loop" behavioral container. Every *Statement* consists of a list of *Instructions*, which form the leaf nodes in the EL model and perform the actual operations.

Furthermore, due to the flexibility of the EL model and the possibility of unbounded in time of rule evaluations in the sense of CASM, we decided to translate EL instances in the HDL back-ends to asynchronous digital designs. Hence, every *Function*, *Intrinsic*, *Statement* etc. from the EL model follows a request-acknowledge handshake protocol. Currently we only focus, besides the software C back-end, for the hardware back-ends on the generation of VHDL code with an assumed annotated timing information. The generated designs are validated in a HDL simulator environment. But in the future the generated code shall be synthesizeable to FPGA boards as well.

Compiler Design

From the software design point of view of the compiler, both presented models (IR and EL) follow a Single Static Assignment (SSA) based internal representation. They use a similar class design and analyze/transformation pass design proposed by the Low Level Virtual Machine (LLVM) compiler infrastructure by Lattner and Adve [89]. The latter was used in early experiments to translate the CASM IR model to the LLVM IR model, but due to the retargetable focus for assembly code it turned out that the LLVM IR model was to low-level to realize our retargetable approach. Therefore, we started the design of the EL model.

2.3 Discussion

The presented model-based transformation approach can have various applications and implications. As one example, the designing and developing of asynchronous logic in the hardware domain is a quite challenging topic. Despite its development several

decades ago and several very beneficial properties asynchronous logic design, which is data driven and runs as fast as possible in all situations, is rarely used nowadays. Reasons are of course its disadvantageous properties such as bad testability but also required sophisticated knowledge for designers and missing tools.

The presented model-based transformation approach in this chapter can be used to tackle the latter points by suggesting a tool or a way to generate multiple circuit implementations from a single description. The aim for hardware communities to convert specifications written in various input languages, e.g. C or VHDL, to an unified representation is the main goal, which can be achieved by creating building blocks (semantic vocabulary) specified through the ASM-based formal method.

The ASM artifact can be used to generate the circuit in the desired (a)synchronous design style. In contrast to the broadly used synchronous approach, which uses a central clock to coordinate the single units, data-driven asynchronous logic utilizes dedicated signals to indicate when new data has arrived (request) and when the old data has been processed (acknowledge). This allows the circuit (1) to work as fast as possible and (2) to adapt much better to Process-Voltage-Temperature (PVT) variations. Where synchronous circuits fail due to timing violations their asynchronous counterparts still deliver correct results.

Despite these argument and being available for several decades now, asynchronous logic is still used only marginally in digital design [109]. Reasons for that are manifold: Since data is processed immediately after arrival, a very high level of concurrency is achieved. This leads however also to a huge amount of possible states and thus bad testability, as all of them have to verified. Furthermore, tool support is still lacking.

At the moment this seems to be a chicken-egg problem: Companies are not ready to develop new tools until the demand is high, which does however not grow due to the lacking tool support. Please note that, albeit with increased effort, it is actually possible to design asynchronous logic with available tools assumed that the designer has sophisticated knowledge about the asynchronous design style, which differs in certain points significantly from the synchronous one.

For example one has to make sure that the communication protocol (request and acknowledge) can be fully executed, which sometimes requires the introduction of additional memory elements [36]. Thus switching to asynchronous logic means retraining the designer which results in high risk for a company.

Generating asynchronous circuits systematically from Data Flow Graph (DFG)s, i.e., models that show how data is propagated from one operation to the next but neglects the inherent timings, is already possible. For this purpose operations and special constructs such as *merge* or *join* are simply replaced by corresponding asynchronous block as shown e.g. in [79]. In computer architectures similar concepts have already been implemented successfully, e.g. in data flow processors [44] or in out-of-order computational units, which essentially generate a DFG at run time.

In summary, we currently have a very good IR for asynchronous logic, namely

DFGs, and various methods to describe the behavior on different abstraction levels (e.g. C, VHDL, or Verilog). The main issue is to properly close the gap between them. Of course, we are aware that available compiler tools, for example LLVM [89], are capable to generate data flow models, which was for example used by Josipović et al. [79] to develop Elastic Pipelines [29] based on a algorithmic description in C.

In general, the latter approach cannot be used to generate the DFG into asynchronous logic, due to the synchronous or computer architectural assumptions that are utilized by the compilers. An example are memory references which are common in software however hard to convert to hardware.

Therefore the biggest challenge currently in our opinion is the generation of an IR that can be used as starting point for multiple purposes, e.g., development of synchronous and asynchronous circuits. Therefore, we propose a tool that can convert circuit descriptions in various formats and abstraction levels, e.g. pseudo code, C, or VHDL, into a single common IR.

The latter is based on the well-known formal method ASM [63] [23] which allows exactly what we need: a unified specification of the circuit behavior independent of the desired implementation details. The latter are only fixed during the export to an implementation specific IR, for example DFG in the case of asynchronous circuits. Overall the task of the desired tool is to (a) extract valuable information from different descriptions and (b) export this information to the desired format and circuit style.

Due to the fact that asynchronous logic is only marginally used at the moment, the actual question is how the proposed tool can change this situation?. Specifying the new IR building blocks (semantic vocabulary) as well as the implementation of various front-ends and back-ends can only happen in steps. One step would be to start with specific front-ends for Verilog and VHDL and, naturally, with an asynchronous circuit style back-end.

In the beginning this would allow designers to read existing (synchronous) circuit descriptions and generate asynchronous counterparts, which gives them the chance to (a) get a quick estimation what asynchronous logic is capable of and thus support future asynchronous implementation and (b) learn by comparison how to properly design asynchronous circuits on the fly. The proposed tool is also supposed to enhance verification and validation as the transformation to a formal model enables the usage of automatic verification methods to proof specific properties and thus show correct behavior.

Since the tool described in this discussion does not exist yet, we would first have to specify the IR building blocks (semantic vocabulary) as first step. To find a proper abstraction, we have to analyze the structure of asynchronous circuits and systems and based on these create suitable data structures that are capable to model the data flow graph appropriately. From that it should be easy to generate asynchronous logic automatically by using the already available gates and building blocks.

2.4 Conclusion

We have outlined our CASM based retargetable compiler infrastructure and the model-based transformation approach which will enable the reuse, integration and execution of a single CASM specification in different software and hardware environments through the usage of the EL model.

The current development status of the compiler and the models are in an early state. Major compiler infrastructure and transformation passes are implemented to parse, dump and transform CASM input specifications. We were able to retarget a small CASM filter application to a valid C program and VHDL digital design (not synthesizeable yet). The example application consists of three functions, one rule and two parallel update terms.

The overall goal we want to achieve in our future work is to create at least for four language domains a translation back-end implementation. CASM specifications shall be translated to C11 (native), Python (script), JavaScript (web) and VHDL (hardware). A possible field of application would then be the construction of a new RISC microprocessor design in CASM like RISC-V[2] [8] or Small-Scale Experimental Machine (SSEM)[3] [35]. The proposed retargetable approach of our modeling system would generate then directly an Instruction Set Simulator (ISS) for software debugging, an ISS for integration in a website (e.g. for presentation and testing purposes), and a valid synthesizeable hardware implementation for e.g. FPGA or ASIC platforms.

Regarding the asynchronous circuits development, they are far less popular than their synchronous counterparts. We have discussed in this chapter a possible road-map of a tool that might change this, as it is capable to generate from a single (high-level) description both synchronous and asynchronous circuits by using the presented model-based transformation approach. For that purpose we need an IR that represents through proper building blocks (semantic vocabulary) parsed input descriptions. This IR is specified using an ASM-based language like CASM.

[2] See `https://riscv.org` for the computer architecture description.

[3] See `https://github.com/casm-lang/libcasm-tc/blob/master/application/SSEM.casm` example specification.

"Programming is not about typing,
it's about thinking." – Rich Hickey

CHAPTER 3

Intermediate Representation

Over the past years, there have been many approaches to implement concrete ASM-based modeling and specification languages. All of those approaches define their type systems and operator semantics differently in their internal representation, which leads to undesired or unexpected behavior during the modeling, the execution, and code generation of such ASM specifications. Moreover, all approaches address specific refinement and optimization needs directly in their compiler infrastructure, which makes reuse of compiler analyses and transformation passes highly unlikely. To address this problem, the core elements of ASM-based languages have to be defined in a reusable and uniform representation. The goal is to enable a language engineer to avoid specifying and implementing core features of ASM languages in a redundant way and focus only on the needs of the language user during the language design and development. In this chapter[1], we present the CASM modeling language's compiler IR, named CASM Intermediate Representation (CASM-IR), which can be seen as the ASM core part of the described compiler design from previous chapter 2 (see Section 2.2). CASM-IR is based on a well-formed ASM-based specification format. The CASM-IR is conceptualized from the ground up to ease the formalization of compiler analysis and transformation passes to create ASM-based optimizations. Based on our CASM-IR implementation, we were able to easily integrate the front-end of our statically inferred CASM modeling language.

3.1 Introduction

ASMs are used to describe formally the evolving of function states in a step-by-step manner. This also explains why ASM theory was formerly called *Evolving Algebra* [64]. Based on the ASM programming language model from Gurevich, several tools with DSL implementations were created to solve application-specific problems, which were summarized by Börger [22]. The diversity of ASM-based applications[2] is widespread,

[1] The content of this chapter is a revised version of the ABZ'18 paper [117].
[2] A list of various ASM applications is depicted at `https://abz-conf.org/method/asm`.

ranging from formal specification semantics of programming languages, such as those for Java by Stärk et al. [146] or VHDL by Sasaki [133], compiler back-end verification by Lezuo [90], software run-time verification by Barnett and Schulte [11], software and hardware architecture modeling e.g. of UPnP by Glässer and Veanes [60], or even RISC designs by Huggins and Campenhout [72].

Despite this diversity in applications, over the past years, different ASM-based language dialect were created to cover single or multiple application specific problem domains. This might not be perceived as a problem, as many *language users* [83] like to choose among multiple language dialects. The problem however is that the *language engineers* [83] craft and design those languages according to the needs of the *language user* and bind their implementations to a specific execution environment technology, instead of generalizing the mathematical foundation of the ASM-based languages in an independent model representation. As a consequences, those languages are difficult to integrate with each other [154], cannot easily be based on a common execution environment technology, and establishing a common set of language tools is difficult.

Moreover, the binding to various execution environment technologies introduces undesired and unexpected behaviors, e.g. if the same algorithm so to say is specified with different ASM modeling languages and the model execution leads to different floating point values or depending on the Integer representation to different overflow states. To overcome this *uniform ASM representation problem* a clear, precise, and formal IR has to be introduced, which has the ability to represent various ASM language constructs of different contexts. A potential of such an ASM-based IR is depicted in Figure 3.1. Due to the decoupling of the ASM syntax from the specific execution environments and target languages other ASM languages can benefit from the same IR as mid-level target in their language implementation and design by reusing language definitions and run-time features. The transformation step from the ASM-based IR to specific targets was sketched in one of our earlier work about a model-based transformation approach to reuse and retarget ASM specifications [123].

The major advantages of such an approach are the generalization of ASM-related analyzes, optimization, and transformation capabilities of compilers – first envisaged by Lezuo et al. [94] – based on a single uniform model. Furthermore, another benefit for existing ASM languages is to directly reuse the numeric as well as the – proposed by

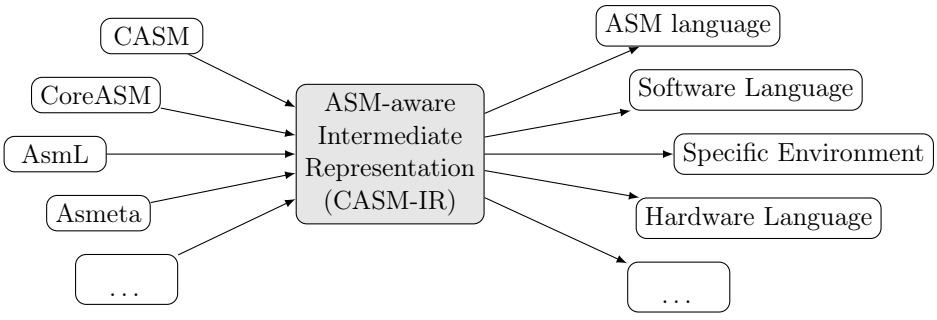


Figure 3.1: Potential of ASM-aware Intermediate Representation

Lezuo [90] – symbolic execution of specified ASM models. A huge disadvantage in the perspective of a *language engineer* is to port existing ASM language implementations to such a uniform ASM model.

This chapter focuses on the design, implementation, and integration of an ASM-based IR model named CASM-IR to address the *uniform ASM representation problem*. The main contribution of this chapter is the definition of a well-formed ASM-based IR model which is independent of language front-ends and provides a well-defined type system, operator and built-in semantics.

This chapter is organized as follows: In Section 3.2 we provide an introduction and background to the theory and formal method of ASMs. Section 3.3 describes the research context and the motivation of this chapter. In Section 3.4 we describe our CASM-IR model. Section 3.5 presents details about the current implementation and integration of the CASM-IR. Section 3.6 gives an overview of the related work regarding IR's of other ASM languages and tools. Finally, in Section 3.8 we conclude the chapter and outline the future work.

3.2 Background

This section provides a small introduction into the ASM theory and discusses relevant properties for understanding this chapter. Readers already familiar with ASMs and their corresponding syntax representations and semantics may consider to skip the whole or some parts of this section.

Accordingly to Gurevich and Tillmann [66], 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 ASM theory and its resulting formal method consists 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; (3) stepwise *refinement* of the reference model by instantiating more and more concrete models which uphold the properties of the reference model [26].

An ASM specification defines a state out of mathematical *functions* with corresponding a type domain mapping or, in short, a type relation. Every function has an arity and if $\vec{z}$ is an n-dimensional vector and f an n-ary function then $\vec{z}$ is called a *location*. In order to change and manipulate the current function state, *rules* are used to define computations and transitions of function location values. This change is specified by an *update* rule, which produces a new update $u = (f(\vec{z}), v)$ consisting of the new value v, an n-ary function f, and a specific location $\vec{z}$. A specified *agent* executes a rule in an atomic step behavior an collects all produces (partial) updates [66] and applies those to the function state. A *named rule*[3] can contain sub-rules

[3]The *named rule* refers to a user defined transaction and computational rule.

the following basic ASM rules: *skip*, *update*, *conditional*, *for-all*, *choose*, *iterate*, and *call*. A *skip* rule does not perform any computation. An *update* rule produces new updates as explained before. The *conditional* rule enables branching and selective rule computations in form of an *if-then-else* block. A *for-all* rule expresses bounded parallelism to be applied to a rule r over a variable x in the form if $\forall x : r(x)$. The *choose* rule introduces non-determinism to ASM by selecting randomly a value of a given set which shall be used for further computation in the underlying rule. Repetitions can be expressed through the *iterate* rule. Last but not least, the *call* rule enables the possibility to invoke sub-rule calls, even recursion is possible to express in ASM specifications.

Each of the above named rules can be assigned to certain execution semantics. By default, ASM uses a *parallel execution semantics*, meaning that all rules inside a rule are executed in a synchronous parallel manner. In 2000 Gurevich [64] extended the ASM computation model with a *sequential execution semantics* allowing to introduce sequential sub-machine steps producing *pseudo updates* [94] to describe sequential computations. This allows to interleave parallel and sequential execution semantics inside a specified *named rule*. As described in [66] the interleaving introduces a complex situation to ASM interpreter and compiler developers, since the *lookup* of a certain *function* at a certain *location* has to respect the surrounding nested parallel and sequential state. This problem was efficiently addressed and solved by Lezuo et al. [94] by introducing a ASM specific *update-set* implementation based on hashing.

An important concept in ASM languages is the *agent*. ASMs have a *single-* and a *multi-agent* concept [63]. The basic ASM specification uses a *single-agent* that executes a given *named rule* and collects the produces *updates*, which are applied to the global *function* state. The latter is called in ASM a *step*.

Multiple *steps* form an ASM *run* and every *run* ends with a specified *termination condition*. To describe and specify concurrent problems, the *multi-agent* concept in ASMs defines three different modes of operations allowing the definition of more than one *agent* executing a *named rule*. The three modes of operations are: (1) *synchronous*, (2) *asynchronous*, and (3) *concurrent*;

A *synchronous* mode of operation can be seen as a lock-step scenario. All *agents* wait as long as all *agent* have performed one *step* and if the termination condition is not fulfilled another *step* is performed. In an *asynchronous* mode of operation, one *agent* gets randomly selected to perform its *step* one after another. The *concurrent* mode of operation is the most generic one and can be seen as a free-run scenario.

All *agents* run independently and execute their assigned *named rule* as long as the termination condition is reached. Börger and Stärk [26] define multiple termination conditions. The two most common termination conditions are: (1) that an ASM run terminates if all *agents* have no valid *named rule* assigned to them and (2) if no *updates* are produced during an ASM step by any *agent* the ASM run terminates.

3.3 Motivation

The broader context of our research is the creation of a modern state-of-the-art ASM modeling language implementation named the CASM, as well as transformation and deployment of CASM specifications to executable artifacts[4]. The primary application context of this work is the specification of embedded systems in a formal way.

However, in the context of CASM, we not merely focus on specific application contexts like embedded systems, but rather aim to describe and specify arbitrary systems and therefore software and/or hardware application applications and components. This overall idea is not new, but our approach to achieve this goal of generic transformations is different from a language engineering perspective, because we set our ASM-based IR into the center of the front-end language development.

Other ASM language approaches, which are described in Section 3.6, do not, because they implement a forward directed transformation from ASM to the desired target language like C or C++. The transformation of ASM source specifications to specific target languages is by no means trivial. It involves the mapping of a mathematical-based specification model to a real executable program, which for itself resides in a specific execution environment.

To overcome this complex transformation, we proposed and followed a model-based transformation approach in our earlier work [123], which defines four abstraction layers (illustrated in Figure 3.2). At the top resides the *ASM Source Modeling Language* layer that includes besides the language grammar definition the lexer, parser, type inference, type checker, and AST representation. A parsed input specification gets translated to the next layer, the *ASM-aware Intermediate Representation* layer. At this abstraction layer the CASM-IR, proposed in this chapter, is defined. It allows us to analyze, transform and optimize the input specification for ASM related properties.

To elaborate and describe the optimization potential of ASM-based languages consider the following *swap* example specification depicted in Listing 3.1 described in CASM language[5]. It defines a *single-agent* execution environment starting at the *named rule* `swap` (Line 1), two null-ary functions with relation %: → Integer$ (Line 3 and Line 4), and the *named rule* `swap` at (Line 6 to Line 11) with a *parallel execution semantics* (denoted by curly braces) containing three *update* rules.

[4]For more information about CASM refer to the project website at `https://casm-lang.org`.
[5]The CASM syntax is described at `https://casm-lang/syntax`.

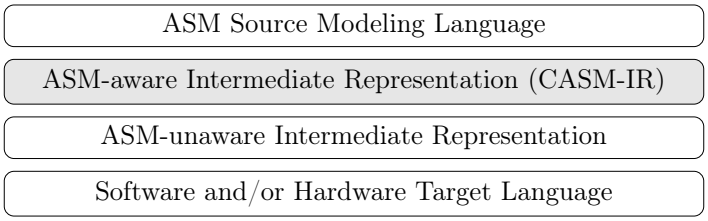


Figure 3.2: CASM Abstraction Layers

An efficient code generator – as proposed by Lezuo et al. [94] – would have generated code as depicted in Listing 3.3. This generated code is a valid C implementation of the input CASM specification for the *swap* example. Still there is a lot of room for possible optimizations, because a closer look and appropriate ASM-based static compiler analysis would detect that in this example the explicit termination condition (see Listing 3.1 at Line 10) reduces this specification to only one ASM *step* without the need to track any ASM-related functional state.

The optimal code for this *swap* example is depicted in Listing 3.2. It would simply transform into an empty C `main` function. So there is a huge optimization potential on the CASM-IR level.

```
1  CASM init swap
2
3  function x : -> Integer
4  function y : -> Integer
5
6  rule swap =
7  {
8    x := y
9    y := x
10   program( self ) := undef
11 }
```

Listing 3.1: Swap Example (CASM)

```
1 int main( int argc, char** argv )
2 {
3     return 0;
4 }
```

Listing 3.2: Swap Example (C, optimal code generation through optimization)

```
1  struct Integer x;
2  struct Integer y;
3  struct Agent program;
4
5  void swap( void )
6  {
7      struct Integer tmp;
8      tmp = x;
9      x = y;
10     y = tmp;
11     program = { 0 };
12 }
13
14 int main( int argc, char** argv )
15 {
16     program = { &swap, 1 };
17     while( program.defined )
18     {
19         (*program.value)();
20     }
21     return 0;
22 }
```

Listing 3.3: Swap Example (C, efficient code generation)

The CASM-IR gets further transformed in the next layer called *ASM-unaware Intermediate Representation* (see Figure 3.2). At this abstraction layer the transformed specification has no longer any knowledge about the semantics or behavior of ASMs. Therefore it can be analyzed, transformed and optimized for traditional properties like execution speed or program size. In the final layer of Figure 3.2, the *ASM-unaware Intermediate Representation* is mapped to various *Software and/or Hardware Target Languages*.

Those CASM system abstraction layers describe a full transformation of an ASM specification to its desired target language. Due to the proposed layered structure, it is possible to only use a sub-set of the full functionality as well. For example, AST-based execution is used to recursively walk over the in-memory AST representation and interpret the input specification as it was parsed.

In the context of AST-based execution, *language engineers* can (re)use the type system from the ASM-aware IR layer and can rely on its defined behavior and semantics. Therefore the CASM-IR can be used not only for transformation and code generation purposes, but also for AST-based interpreter applications.

Furthermore, besides ASM-based languages, this proposed IR and its functionality could be used for other functional programming languages as well for their function definitions, type relations, numeric and symbolic computations. The approach to address the *uniform ASM representation problem* – introduced and described in Section 3.1 – with the CASM-IR raises several concerns regarding its existence and usefulness.

First of all, the effort to investigate into such an IR design arises from the fact that accordingly to the state-of-the-art and to our knowledge no comparable IR for ASM languages with the focus on well-formed, reusable, retargetable, and optimizable ASM specifications exist yet.

Second, as presented by Lezuo et al. [94], the optimization potential is huge of ASM languages regarding redundancy eliminations, but still not covered and addressed by any ASM language implementation in a unified manner.

3.4 CASM-IR

This section describes our ASM-based IR design that can be (re)used for designing and building ASM-based and other possible functional related specification languages. Before we go into the details of the model and the format of this IR, we first outline the composition of our CASM system [123].

Figure 3.3 depicts a more detailed overview of the sketched abstraction layers from Section 3.3 (see Figure 3.2). A parsed ASM source – in our case the CASM language – gets translated to an AST representation and necessary type information gets inferred. In order to do so, the CASM-IR – depicted as Model (IR) – needs to provide type information for all possible operators and their type relations, which a language front-end can use, to implement a type inference pass. Furthermore, the

CASM-IR model provides the ability to directly implement AST-based interpreter applications on top of it, because language front-ends can access the implemented run-time of the IR to evaluate expressions and terms.

If the execution shall be done directly using the IR model itself, a language front-end just has to perform a model-to-model transformation from its AST-based representation to an instance of this IR model. At this point the IR can optimize the specification for ASM-related properties fully decoupled from the original input specification in form of an AST representation. Some optimization properties were proposed by Lezuo et al. [94]. Furthermore, then the IR instance can be executed by the run-time implementation of the IR model.

For further processing (code generation) of the specification to a specific programming target language, the IR instance can be transformed into an EL model, as proposed in our earlier work [123]. The EL model is another compiler IR, which allows to express the CASM run-time in a target independent way.

Furthermore, the CASM-IR instance of an input specification gets transformed and combined on EL level and enables ASM-unaware compiler optimizations for properties like execution speed or final target program size. In the final step, the transformed EL instance gets combined with the run-time from the EL itself and transformed into the desired target language or execution environment.

As proposed in [123] the EL model needs to be designed and evolved that allows generic targeting ranging from high-level software (e.g. C, C++, JavaScript), low-level software (LLVM [89], GENERIC/GIMPLE [104], Assembler), high-level hardware (VHDL, Verilog, SystemVerilog), to low-level hardware (FIRRTL [73], Yosys [159], Register Transfer Level (RTL) netlists).

Motivating Example

To better understand the solving of the research question regarding the *uniform ASM representation* problem that CASM-IR deals with, we describe a small ASM specification and point out the issues, which are addressed by the CASM-IR design

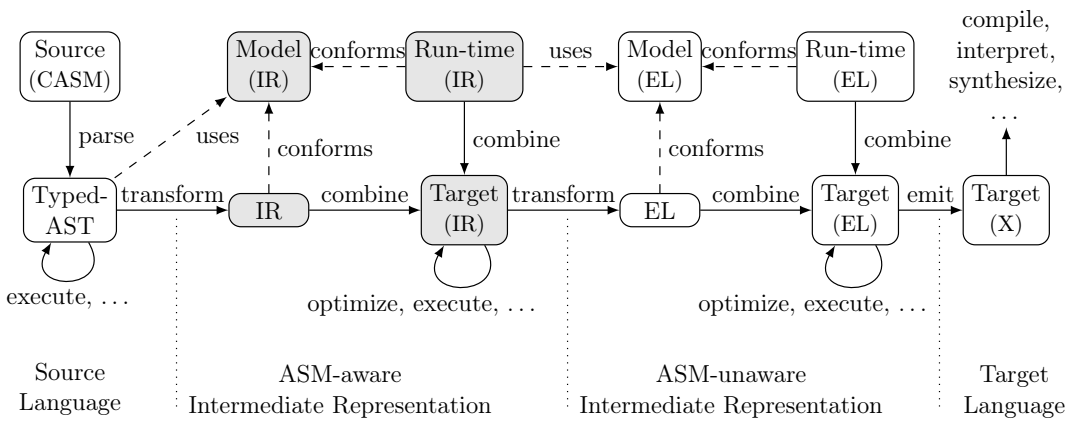


Figure 3.3: CASM System Design (High-Level Overview)

```
CASM init swap

function x : -> Integer
function y : -> Integer

rule swap =
{
  x := y
  y := x
  program( self ) := undef
}
```

Listing 3.4: Swap Example (CASM)

```
var x as Integer
var y as Integer

swap()
  x := y
  y := x

Main()
    swap()
  step
    // terminates after this step
```

Listing 3.5: Swap Example (AsmL)

```
CoreASM swap
use StandardPlugins
init swap

function x : -> Integer
function y : -> Integer

rule swap =
  par
    x := y
    y := x
    program( self ) := undef
  endpar
```

Listing 3.6: Swap Example (CoreASM)

```
asm swap
import ../STDL/StandardLibrary

signature:
  dynamic controlled x : Integer
  dynamic controlled y : Integer

definitions:
  main rule swap =
    par
      x := y
      y := x
    endpar
```

Listing 3.7: Swap Example (Asmeta)

and implementation. Listing 3.4 depicts a valid (high-level) CASM specification of a modeled *swap* algorithm[6]. It defines a rule `swap` (Line 6) and two nullary functions `x` (Line 3) and `y` (Line 4) of result type integer. The `init` (Line 1) defines a single execution agent with starting top-level rule `swap`. Rule `swap` defines a parallel block rule (Line 7 to Line 11) and three update rules.

The first two update rules in Listing 3.4 (Line 8 and Line 9) are producing updates to swap the function values from `x` and `y`. In the last update rule (Line 10), the ASM *program* function gets updated with an undefined value, which results into a termination of the specification, because the single execution agent top-level rule gets set to an undefined value and therefore the ASM execution concludes the model execution.

To get a feel for the *swap* algorithm ASM specification in other ASM languages, we depict three further examples of the same algorithm modeled in *AsmL* [65] (Listing 3.5), *CoreASM* [50] (Listing 3.6), and *Asmeta* [58] (Listing 3.7).

Even in this small specification, several behaviors and definitions are implicit and slightly different in the various ASM languages. E.g. the used function `program` (Listing 3.4 at Line 10) is not an explicitly defined function in this valid CASM specification, because this function definition is hidden from the *language user* and it gets implicitly defined, because it depends on an agent type domain. The type relation of this function would be a projection of the current agent type domain to a stored top-level rule, which is similar in the *CoreASM* specification (Listing 3.6 at Line 12).

Furthermore, the initialization of this `program` function to the rule `swap` is implicit as well. In CASM and *CoreASM* this is achieved by setting the underlying agent through the `init` definition (Listing 3.4 at Line 1, and Listing 3.6 at Line 3). Similar behavior can be achieved in *Asmeta* by setting a certain rule to a `main` rule (Listing 3.7 at Line 9) or in *AsmL* which forces the uses to define a `Main()` rule (Listing 3.5 at Line 8) which controls the computation directly.

Moreover, it can be observed that the *swap* examples of CASM and *CoreASM* explicitly define the termination of the specification whereas the *swap* examples written in *AsmL* and *Asmeta* do not. In order to implement e.g. an AST-based interpreter to execute this specifications a *language engineer* would have to implement a run-time kernel, which handles those implicit defined behaviors.

Furthermore, if we think about optimizing such specifications, implicitly defined behaviors cannot be optimized by transformation passes in a generic way.

Generally speaking we have discovered two implicit behaviors – *initialization of functions* and *agent life cycle handling*. Latter is very important if we consider synchronous and asynchronous multi-agent ASM specifications. To express ASM specifications in a well-formed IR we present in the following sub-sections the definition of our CASM-IR model and its textual representation.

[6] A detailed explanation of the CASM *swap* example is given in Section 3.3.

Type System

Due to the fact that every ASM-based language will eventually be executed by a real machine a term, expression or even a value will have a concrete type. Even Gurevich [64] suggested his ASM language definition lacks explicit typing, and it would be more practical to introduce such. Therefore, in the center of our CASM-IR model stands the type system with all of its possible type domains, which we will call from now on just types[7].

An overview is depicted in Figure 3.4. We can observe that the type system defines very basic (*Primitive*) types like *Boolean* or *Integer* up to very abstract ones (*Template*) like *List* or *File*.

Notable to mention in contrast to other ASM languages is that CASM-IR always tries to be as close as possible to the mathematical foundation of a type. This means e.g. the *Integer* representation is represented as an arbitrary precise Integer with range $]-\infty, \infty[$. There is even the possibility – similar to the Ada programming language – to restrict the type to a certain sub-range. Furthermore, CASM-IR introduces a *Binary* type which can be used to represent any binary bit-precise value with defined bit-size. Along with this type CASM-IR features a set of *Binary* built-in arithmetic operations (see Section 3.4). In the implementation of Lezuo et al. [94] this type was indirectly specified with Integer types by limiting built-in operations over a predefined bit-size values and the language itself was not aware of these operations. Another novel type in CASM-IR compared to other languages is that it features a typed *Reference* type. All references to rules, functions, and derived functions have to be typed to ensure type safety for indirect calls.

Constant Values

Due to the mathematical foundation of ASMs, all typed CASM-IR constants[8] can have besides the type-specified (domain) content, an undefined value. Furthermore, we directly include in CASM-IR the notion of symbolic values that enable a clear

[7]The CASM-IR type description is given at `https://casm-lang.org/ir/types`.
[8]The CASM-IR constant definition is given at `https://casm-lang.org/ir/constants`.

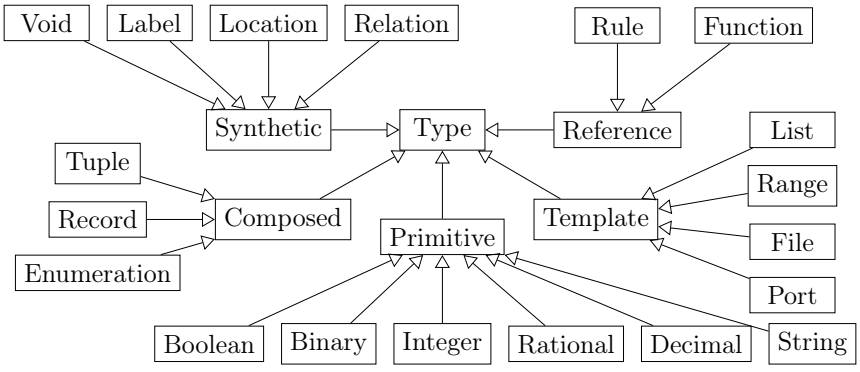


Figure 3.4: CASM-IR Type System (Inheritance Tree)

definition of numeric as well as symbolic execution, whereas the symbolic values are its own domain value as suggested by Lezuo [90].

Function States

States are modeled through the *function* definitions[9]. As defined in [26] every ASM function has a name and an arbitrary type relation. By default every function – accordingly to the ASM definition – is undefined over its type relation domain and needs to be explicitly initialized in CASM-IR. Listing 3.8 depicts a constant `@c0` of type *Integer* and value 123, a constant `@c1` of type *Rule Reference* with relation $:\rightarrow Void$ and an *undefined* value, and a function `foo` with relation $: Boolean * Rational \rightarrow Integer$.

Agents

ASM specifications can either be *single* or *multi* execution agent-based systems [63]. Therefore we provide a model instance to declare only the agent type domain that directly results in the desired behavior. For instance, if we would define the agent type domain to a *Boolean* type, we would define two operational agents. The agent type domain has an important role in the execution of all ASM specifications because starting from a defined agent rule the nested rules get called and so on. Furthermore, the defined agent domain is also used in a special internal *function* named `program` to store the current agent top-level rule as a rule reference.

Listing 3.9 depicts how to set the model instance of the current agent type domain. In Line 2 an enumeration type `bar` gets defined with enumerators `A`, `B`, and `C`, and in Line 5 the agent type domain gets set to the type `bar`. Therefore, we have specified in this example a multi-agent ASM with three agents. As already mentioned in Section 3.4 there is a special function named `program` that controls the execution of the agents in its kernel of ASM specifications, which heavily depends on the agent type domain. Line 9 shows the corresponding `program` function definition with the agent type domain `bar`. Furthermore, this shows clearly the explicit definition of the special internal function named `program` in CASM-IR compared to the indirect definition in the CASM high-level specification from the Section 3.4.

[9]The CASM-IR function definition is given at `https://casm-lang.org/ir/functions`.

```
1 ;; integer constant '123'
2 @c0 = i 123
3 ;; 'undefined' rule reference
4 ;;  constant of relation : -> Void
5 @c1 = r< -> v > undef
6 ;; function definition 'foo'
7 ;; with relation:
8 ;; Boolean * Rational -> Integer
9 @foo = < b * q -> i >
```

Listing 3.8: Constants and Functions

```
1 ;; enumeration type definition
2 bar = { A, B, C }
3 ;; setting agent type domain
4 ;; to enumeration type 'bar'
5 .agent = bar
6 ;; function definition 'program'
7 ;; with relation:
8 ;; bar -> RuleRef< -> Void >
9 @program = < bar -> r< -> v > >
```

Listing 3.9: Enum. and Agents

Rules

The actual computation in ASMs is specified through transition rules and CASM-IR does contain a notation of rules as well, but only for the *named rule* definitions. Other ASM rules like *update*, *conditional*, *for-all*, *choose*, *call* etc. are represented in CASM-IR through nested blocks and instructions (see Section 3.4).

Derived Functions

Another important specification component in CASM-IR are *derived* functions or *deriveds* for short. It can be seen as a kind of typed macro to reuse *state-less* or *side-effect free* calculations. This means, that in *deriveds*, no state changes are allowed to be performed; ergo, no *Update* rules are allowed in derived function definitions.

Built-in Functions

The CASM-IR features several built-in functions[10]. Depending on the behavior they act like *deriveds* or *rules*. Since CASM is a statically typed language and it does not allow implicit type casts, all of the required casting facilities are directly specified at IR level. Besides basic casting facilities, a set of arithmetic operations is defined for the *Binary* type at built-in level inside the IR as well. Figure 3.5 depicts the definition of a *Boolean* to *Binary* of bit-size b cast definition and a Zero-Extension (`zext`) definition by resizing a *Binary* type of bit-size n to a bit-size of b.

Blocks, Instructions, and Registers

All basic expressions and state-modifying rules are represented in CASM-IR as *Instructions* in a SSA form. So produced results of instructions are stored in registers and the type is directly yielded from the specified instruction. This conceptual idea

[10]The CASM-IR built-in definition is given at `https://casm-lang.org/ir/builtins`.

$$asBinary : Boolean * Integer \rightarrow Binary(b), b \neq undef \wedge b > 0$$

$$asBinary(a, b) = \begin{cases} undef & \text{if } a = undef \\ sym' & \text{if } a = sym \\ 0 & \text{if } a = false \\ 1 & \text{if } a = true \end{cases}$$

$$zext : Binary(n) * Integer \rightarrow Binary(b), b > n$$

$$zext(a, b) = \begin{cases} undef & \text{if } (a = undef) \veebar (b = undef) \\ sym' & \text{if } (a = sym) \vee (b = sym) \\ a & \text{if } otherwise \end{cases}$$

Figure 3.5: CASM-IR Built-in Function Definitions (excerpt)

is borrowed from the LLVM compiler IR design by Lattner and Adve [89]. So any instruction call can be specified by a resulting unique register name, an instruction name and possible instruction operands with explicit types. This also indicates that the CASM-IR follows a register machine design and implementation approach.

Basic ASM rules like *skip*, *choose*, or the definition of execution semantics (*fork* and *merge*) are represented as single instructions. Novel in CASM-IR is that it explicitly models the reading (*lookup*) and writing (*update*) of ASM function states by dedicated instructions. This allows to analyze and optimize CASM-IR specifications as suggested by Lezuo et al. [94].

A *location* instruction performs the function location calculation. How the location is calculated is not fixed and has to be decided in the run-time implementation. E.g. a common technique would be the calculation of a function location by a certain hashing algorithm.

The *lookup* instruction determines at a certain point in the specification, which state value is assigned to a certain function depending on the nested parallel and sequential execution semantics. The argument needed to perform a lookup is a location constant.

An *update* instruction produces a new *location* and *value* pair, which gets applied to the surrounding (local) function state also known as *pseudo state* [94]. Therefore, an update instruction needs, besides the exact calculated function location, a value operand.

Listing 3.10 depicts an example usage of the *location*, *lookup*, and *update* instruction. In Line 2 a *location* calculation is performed for the function `foo` which has accordingly to the type one Integer argument. At Line 3 the actual *lookup* of the function value is performed. And in Line 5, a new *update* is performed to the same location were the lookup was performed.

Similar to traditional assembler languages, the CASM-IR includes a *call instruction* as well, but this call instruction is used for multiple invocation types. It is used to call specified rules, derived functions, and pre-defined built-ins either directly by its name or indirectly through a register value of a reference type. Besides the generic *call instruction* there exist several instructions to perform intermediate calculations of arithmetic, logical, and comparison operations[11]. Figure 3.7 depicts the definition of the addition (`add`) and the equal (`equ`) instruction.

[11]The CASM-IR instruction definition is given at `https://casm-lang.org/ir/instructions`.

```
1 %r0 = ;; ... calculation which yields result of type 'i'
2 %r1 = location < i -> i> @foo, i %r0 ;; yields type 'loc'
3 %r2 = lookup loc %r1 ;; yields type 'i'
4 %r3 = ;; ... calculation which yields result of type 'i' and uses '%r2'
5 update loc %r1, i %r3 ;; produces an update to function 'foo'
```

Listing 3.10: Location-, Lookup-, and Update-Instruction

Multiple instructions are compound to a Statement Block (SB) whereas the execution semantics of the instructions is always sequential. Several blocks are grouped together and form an Execution Semantics Block (ESB) which can either have a *parallel* or *sequential* execution semantics. Additionally, every ESB contains, besides the sub-blocks, an *entry* and an *exit* SB, in which the actual execution semantics is specified by appropriate *fork* and *merge* instructions. Figure 3.6 depicts the composition of rules, the ESB and SB blocks as well as instructions.

Motivating Swap Example in CASM-IR

In this section we present an example output of the transformed motivating example `swap` CASM specification from Listing 3.4 to our CASM-IR. The performed model-to-model transformation is implemented in the CASM front-end (see Section 3.5). It shall summarize several of the presented concepts and sketch some optimization possibilities, which can be obtained through the representation of ASM specifications in the CASM-IR. Note that this presented transformed motivated example is valid for the other presented ASM `swap` specifications as well (Listing 3.5, Listing 3.6, and Listing 3.7).

Listing 3.11 visualizes a CASM-IR instance, where the missing definitions and implicit behaviors from Listing 3.4 are explicitly specified. In the transformed specifi-

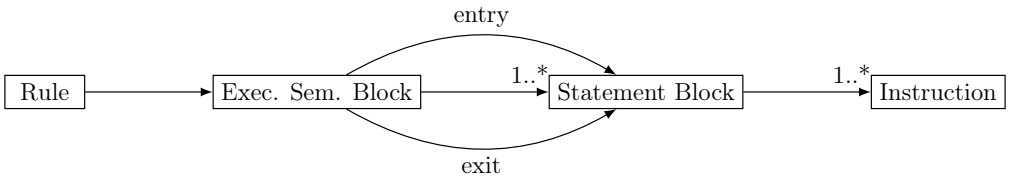


Figure 3.6: CASM-IR Rules, Blocks, and Instructions

$$add : Integer * Integer \rightarrow Integer$$

$$add(a,b) = \begin{cases} undef & \text{if } (a = undef) \vee (b = undef) \\ sym' & \text{if } ((a \neq undef) \wedge (b = sym)) \\ & \veebar((a = sym) \wedge (b \neq undef)) \\ a + b & \text{if } otherwise \end{cases}$$

$$equ : Type * Type \rightarrow Boolean$$

$$equ(a,b) = \begin{cases} true & \text{if } (a = undef) \wedge (b = undef) \\ false & \text{if } ((a = undef) \wedge (b \neq sym)) \\ & \veebar((a \neq sym) \wedge (b = undef)) \\ sym' & \text{if } (a = sym) \vee (b = sym) \\ a = b & \text{if } otherwise \end{cases}$$

Figure 3.7: CASM-IR Instruction Definition (excerpt)

cation we can observe that first of all the agent type domain gets set to a enumeration type named `a` (Line 3) with the name `$` (Line 2). This means that the agent type domain consists of only one concrete value and hence we have a single execution agent ASM specification. Thereafter, a forward declaration of the rule `swap` is specified (Line 4) because the next listed constants (Line 5-8) contain the symbol of the `swap` rule to define a rule reference constant. Next, three functions are defined. The `program` function (Line 9) with the previous defined agent type domain that stores the ASM agent top-level rule reference. After that the functions `x` (Line 10) and `y` (Line 11) are defined accordingly to the originally input specification. Before the definition of the `swap` rule gets defined, the initialization of the ASM state has to be specified, which at least has to set the correct starting rule of the agents. Note that all function states in ASMs are by default undefined. Last but not least the rule `swap` gets defined. It contains a parallel execution semantics block with three trivial statements and several location, lookup and update instructions.

Regarding the optimization potential in this revised example we can detect several possible ASM-related optimizations. The most obvious one would be a hoisting optimization of redundant location calculations, because the location of nullary functions will always be the same. The calculation e.g. of the location of function `y` at register `%r1` (Line 28) could be moved up before the fork instruction of the entry section at `lbl2` (Line 24). And the location calculation of function `y` at the register `%r6` (Line 36) can be removed and all its uses can be replaced by `%r1`. The same applies for the location of function `x` and register `%r3` and `%r4` (Line 30, Line 34).

Figure 3.8 depicts an excerpt of the AST from Listing 3.4 were the focus set on the *update* rules. By comparing this representation with the CASM-IR in Listing

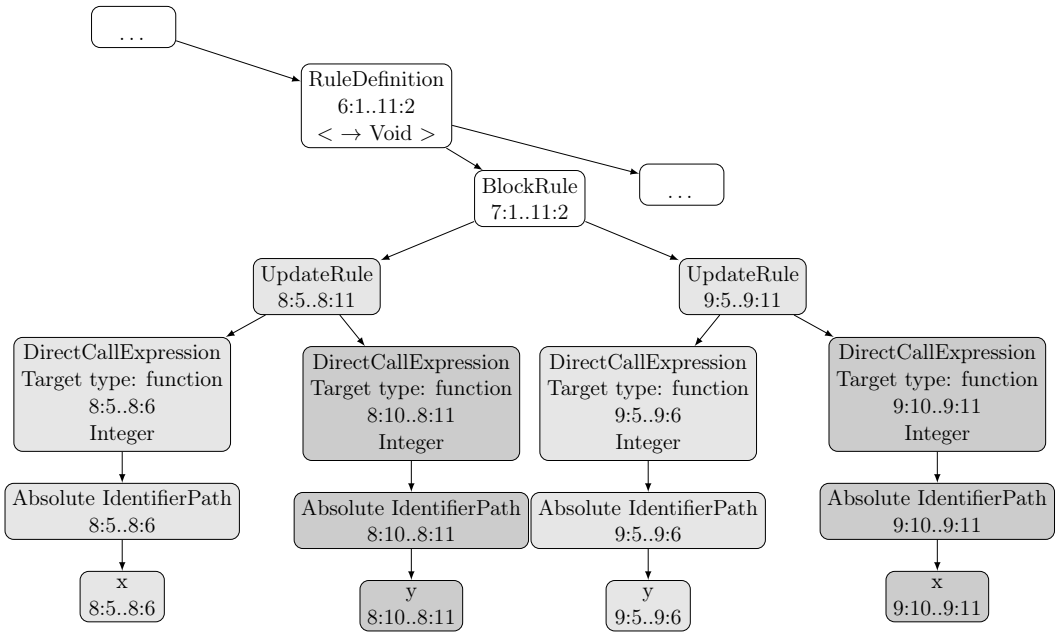


Figure 3.8: Swap Example (AST, excerpt)

3.11, we can detect, that every *lookup* and *location* calculation is explicitly stated in the CASM-IR representation whereas in the parsed CASM specification AST representation only the function symbol names `x` and `y` appear on the left-hand side (*location* of the *update*) and right-hand side (*location* and implicit *lookup*) of an *update* node.

3.5 Implementation

Figure 3.9 depicts the CASM system implementation libraries visualized as a library dependency graph. The CASM run-time and back-end libraries are based on corresponding CASM unaware Just-in-time Emitting Language (CJEL) libraries (situated one layer below the CASM libraries). The CJEL is the concrete implementation of

```
1  CASM-IR                                ;; CASM-IR specification header
2  a = { $ }                              ;; definition of enum. type 'a'
3  .agent = a                             ;; set agent type domain to type 'a'
4  @swap < -> v>                          ;; declaration of rule 'swap'
5  @c0 = r< -> v> @swap                   ;; 'swap' rule reference
6  @c1 = a $                              ;; agent constant of single agent
7  @c2 = r< -> v> undef                   ;; undefined rule reference
8  @c3 = a $                              ;; agent constant of single agent
9  @program = <a -> r< -> v>>             ;; 'program' function definition
10 @x = < -> i>                           ;; definition of function 'x'
11 @y = < -> i>                           ;; definition of function 'y'
12 @init -> v = {                         ;; definition of 'init' rule
13   lbl0: entry                          ;; ESB entry block of lbl0
14     fork par                           ;; fork instruction parallel
15
16   lbl1: %lbl0                          ;; SB lbl1 in ESB lbl0
17     %r0 = location <a -> r< -> v>> @program, a @c1
18     update loc %r0, r< -> v> @c0
19
20   exit: %lbl0                          ;; ESB exit block of lbl0
21     merge par                          ;; merge instruction parallel
22 }
23 @swap -> v = {                         ;; definition of 'swap' rule
24   lbl2: entry                          ;; ESB entry block of lbl2
25     fork par                           ;; fork instruction parallel
26
27   lbl3: %lbl2                          ;; SB lbl3 in ESB lbl2
28     %r1 = location < -> i> @y
29     %r2 = lookup loc %r1               ;; lookup of function 'y'
30     %r3 = location < -> i> @x
31     update loc %r3, i %r2              ;; update of function 'x'
32
33   lbl4: %lbl2                          ;; SB lbl4 in ESB lbl2
34     %r4 = location < -> i> @x
35     %r5 = lookup loc %r4               ;; lookup of function 'x'
36     %r6 = location < -> i> @y
37     update loc %r6, i %r5              ;; update of function 'y'
38
39   lbl5: %lbl2
40     %r7 = location <a -> r< -> v>> @program, a @c3
41     update loc %r7, r< -> v> @c2
42
43   exit: %lbl2                          ;; ESB exit block of lbl2
44     merge par                          ;; merge instruction parallel
45 }
```

Listing 3.11: Swap Example (CASM-IR)

the EL model, which is explained in Section 3.4. All libraries are implemented in the C++11/14 standard.

The implementation of the CASM-IR model consists of two major base classes - *Type* and *Value*. The type system and type hierarchy is implemented according to the definition presented in Section 3.4. All other model instances are sub-classes of the *Value* class. This design approach was borrowed again from the LLVM compiler project where *everything is a value* [89].

Furthermore, every value has a type. The CASM-IR implementation provides a rich Application Programming Interface (API) to provide certain information to front-end implementations. To be more precise here, for every instruction and built-in, it is possible to fetch all defined type relations through an internal type map structure. This enables a clean separation between a front-end language definition and the IR internals.

Based on the CASM-IR, we have designed and implemented our CASM language front-end. Compared to the CASM language implementations from Lezuo et al. [94] the AST has resulted in a much simpler and clearer design then before, because all type, operator, and built-in design decisions were already made in the CASM-IR implementation. Therefore the AST only focuses on the input language itself.

CASM is a statically strong inferred typed language. Hence, the difference between the front-end CASM input specification language and the CASM-IR model is that the front-end language requires a symbol resolver, type checker and type inference pass to fully type the parsed input specification AST representation. In the analyzer passes we use the provided API of the CASM-IR to query and check if certain types, built-ins, and operators (e.g. arithmetic instructions) exist.

Furthermore, during type inference, the front-end can infer the correct type through the pre-defined type relations of the specified CASM-IR operators. E.g. if a type is not possible to be inferred in the front-end, the possible types can be retrieved from the CASM-IR and used as helpful debugging information for language users.

Besides type inference and other analyzes done by the front-end implementation, the most important benefit of targeting the CASM-IR is that a language front-end engineer can directly call evaluation instrumentation functions of the CASM-IR to perform calculations of operator instructions and built-ins.

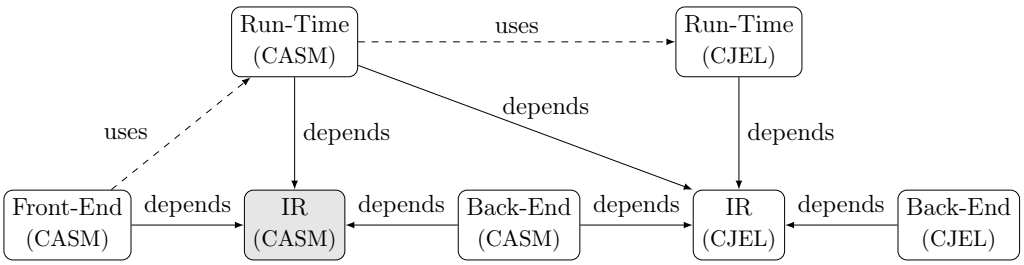


Figure 3.9: CASM System Implementation (Library Dependency Graph)

3.6 Related Work

One of the best-known ASM implementations is the *Asmeta*[12] tool-set with the *AsmetaL* language [58]. The core of *Asmeta* is designed and implemented using the EMF *Ecore* meta-model[13]. Based on the *Ecore* meta-model, the ASM language model of *Asmeta* is directly described as an instance (model). Therefore, the execution and precise calculation of the implemented ASM simulator is bound to the run-time implementation of the *Ecore* meta-model and its EMFs Java interface realizations.

Another notable ASM design and implementation is *CoreASM*[14] originally developed by Farahbod et al. [50]. The focus of CoreASM is to provide a flexible and extensible ASM implementation and to be as near as possible to the described ASM method by Börger and Stärk [26]. CoreASM is implemented in Java and its IR and run-time is directly bound to the JVM.

Microsoft research designed and implemented an ASM language named *AsmL*[15] [65]. AsmL is implemented and based to the `.NET` framework.

Besides CASM-IR, which solves a uniform ASM IR to be language front-end independent, Arcaini et al. [5] proposed a UASM language syntax. Their approach is to unify the front-end ASM syntax representation and this is in the perspective of CASM-IR yet another ASM front-end input specification.

The XASM language[16] by Anlauff [3] compiles the XASM input specifications to C. Ouimet and Lundqvist [113] presented another ASM language named TASM. Their language and simulator approach focused on a real-time ASM notation.

Lezuo et al. [93] introduced in 2013 the CASM language. The origin of this language was that all the (publicly available) existing ASM tools were impracticable for industrial sized applications [91]. The tool-chain presented by Lezuo et al. [94] focuses like the other ASM designs only on the input specification itself, thus those research results were not directly usable by other ASM-based language frameworks. The latter motivated, as already stated in Section 3.3, to rethink the proposed ASM language engineering designs, leading to our model-based transformation approach [123] for the CASM language[17].

Different representation and transformation approaches have been investigated in the *AsmGofer* language by Schmid [136], which is based on the programming language Gofer (similar to Haskell), and the *ASM Workbench* with the *ASM-SL* language introduced by Del Castillo [43], which is implemented in Standard ML.

The *ASM-SL* has been explored further by Schmid [135] to represent and encode specifications in C++. The translation (compilation) scheme was limited to a *double buffering* concept and therefore unable to encode mixing sequential and parallel rules.

[12] See `https://asmeta.github.io` for the Asmeta project.
[13] See `http://eclipse.org/modeling/emf` for the EMF project.
[14] See `https://github.com/coreasm` for the CoreASM open-source project.
[15] See `http://asml.codeplex.com` for the AsmL documentation and project.
[16] See `http://sourceforge.net/projects/xasm` for the XASM documentation.
[17] See `https://github.com/casm-lang` for the CASM open-source project.

CASM-IR solves this by using block-level nested *fork* and *merge* instructions to control the update-set behavior.

Another transformation scheme for ASMs was presented by Bonfanti et al. [20] to represent and encode *AsmetaL* specifications in C++ code targeting Arduino platforms. Their code generator directly converts the ASM specification to the desired target language and run-time environment. By targeting a different run-time environment, platform, or architecture the encoded and implement ASM behavior would have to be re-implemented in every code generator.

Important to point out is that CASM-IR tries to establish a mid-end IR for ASM-based languages similar to the approach for classical programming language IR models such as GCCs *GENERIC* and *GIMPLE* by Merrill [104] or the LLVM IR by Lattner and Adve [89]. All three compiler IR models are independent from the front-end and back-end were they are used and provide an infrastructure for front-end language engineers to target it and an interface for back-end implementer to retarget the IR to different assembler machine code for various computer architectures. The difference to our work is that the abstraction level of those IRs focus more on procedural languages like C whereas concepts like undefined and symbolic states are not addressed and handled at all.

3.7 Discussion

Multiple benefits can be observed in the application of our approach to use and transform to an IR based on ASMs. First of all, due to the fact that all of the basic ASM behavioral elements are specified and implemented in the CASM-IR, an ASM-based language front-end engineer can focus on the high-level language design and directly reuse the implemented run-time and type system. The latter is crucial because the design and implementation of a correct and proper type system for a specification and programming language is complex and time consuming. Second, the transformation to the CASM-IR enables (re)use of the interpreter, compiler, and debugging mechanisms, which are already specified, implemented, tested, and validated in one place of the language tool-chain. This goes hand in hand with introducing new complex or syntactic sugar syntax elements that can be mapped to CASM-IR primitives; that is, it is much easier to support experimentation with new syntax elements which is especially useful for early language design and improvement. Last but not least, by using CASM-IR as a target any supported ASM-based optimization can directly be applied and used in the technology in question.

The first attempt towards an ASM-based IR was presented by Lezuo et al. [94] with a *Par/Seq CFG* representation, which has two major problems compared to the CASM-IR. It does only distinguish between *lookup* and *update* operations including indirectly the *location* calculation. A location pre-calculation optimization is not possible to implement. Furthermore, the CFG is used for analyze purposes only and

if an ASM optimization wants to alter the specification it required huge effort to implement correct AST modifications based on the CFG information [94].

A major drawback of the proposed CASM-IR is that in order to transform to this IR, you need a fully-typed specification or program because of the well-formed and statically typed nature of the CASM-IR. It requires even for other ASM-based languages an implementation of proper type inference passes, to correctly resolve the type relation of expressions, functions, etc. Of course, for highly dynamically typed languages, proper Just-in-Time (JiT) compilation techniques can still (re)use the CASM-IR to perform partial evaluation.

We believe our CASM-IR approach to be generalizable across many different ASM language syntaxes. Rather likely our approach can also be applied for other behavioral formalisms. Both claims have would however requires further studies in future work to be tested.

3.8 Conclusion

We have presented in this chapter CASM-IR, a statically and strongly typed, well-formed ASM-based IR, to provide the ability for ASM-based language engineers to specify the internals of their ASM language in a well-defined representation model. Besides the type system, agent, functions, deriveds, rules, blocks, and instruction semantics, we discussed ASM properties, which are indirectly represented in ASM source languages and made explicitly and typed in the CASM-IR. There are several other issues regarding implicit behavior in ASM-based high-level languages we could point out, but it would go beyond of the scope of this chapter. We have given a short overview of our implementation, corresponding libraries, and discussed the usefulness of our approach.

Regarding the CASM-IR itself, there is a lot of future work in the direction of the type system. The providing of types like *trees*, *sets*, *bags*, and so on, is still an open topic. One solution could be the introduction of a proper type abstraction mechanism inside the CASM-IR itself. We are already working on the implementation, formal definition and verification of ASM-related optimization transformations based on the gained knowledge from Lezuo et al. [94] for our CASM-IR. Another research direction we are working on is the byte-code representation of the CASM-IR. This would allow the implementation of very compact virtual machines for ASM-based specifications.

"Debugging is like being the detective in a crime movie where you are also the murderer." – Filipe Fortes

CHAPTER 4

Concolic Execution

ASMs are a well-known state based formal method to describe systems at a very high level and can be executed either through a concrete or symbolic interpretation. By symbolically executing an ASM specification, certain properties can be checked by transforming the described ASM into a suitable input for model checkers or Automated Theorem Provers (ATP) tools. Due to the rather fast increasing state space, model checking and ATP solutions can lead to inefficient implementations of symbolic execution. More efficient state space and execution performance can be achieved by using a concolic execution approach. In this chapter[1], we describe an improved concolic execution implementation for the CASM language which is based on the previous introduced CASM-IR in Chapter 3. We outline the transformation of a symbolically executed ASM specification to a single TPTP format. Furthermore, we introduce a compiler analysis to promote concrete ASM functions into symbolic ones in order to obtain symbolic consistency.

4.1 Introduction

Due to the mathematical foundation of the ASM theory [63] [25], ASM specifications can be evaluated through either concrete or symbolic interpretation.

All available ASM implementations offer a concrete execution, and some ASM implementations provide a symbolic execution based on model checking (e.g. Farahbod et al. [50] for *CoreASM*). Besides the approaches targeting model checking applications, some ASM implementations transform the specifications into ATP problems to check with off-the-shelve solver tools desired properties (e.g. Arcaini et al. [6] for *AsmetaL* with Satisfiability Modulo Theories (SMT) solver Yices).

A major disadvantage of such techniques is that for rather small ASM specifications, huge ATP input problems are generated which result into large states and long evaluation times of the underlying solver. To overcome this problem, a *concolic*

[1]The content of this chapter is a revised version of the ABZ'21 paper [118].

execution [10] can be used to reduce the number of symbolic path conditions by performing a mixed concrete and symbolic interpretation. Branches inside an evaluation are driven by concrete results and only symbolic states of interest are tracked in the output trace which directly optimizes the results. Therefore, concolic execution [10] trades completeness for computation speed. So far, only Lezuo [90] described a concolic execution approach for ASM specifications.

Based on a prototype version of the CASM language[2] [94], the described concolic execution performed a model-to-text transformation by emitting directly multiple TPTP [149] traces of the symbolically executed specification. A downside of Lezuos' [90] approach is that for each conditional rule (path condition) the generated TPTP trace gets forked into an *if-then* and *else* part resulting into two TPTP specifications which are emitted during the symbolic execution of an ASM specification.

Listing 4.1 depicts an example CASM specification consisting of two functions – `x` and `y` – and a named rule `test` with a block rule, conditional rule, skip rule, and two update rules. This specification represents the running example which was used by Lezuo [90] to describe a *division-by-zero-free* ASM specification expressed in the latest CASM language syntax. Both functions – `x` and `y` – are set explicitly to `symbolic` in order to determine a TPTP trace showing that the function `y` gets only updated with a non-zero Integer value of function `x`.

Two TPTP traces are generated by using Lezuos' [90] implemented (closed source) symbolic execution. Listing 4.2 depicts the *if-then* part and the Listing 4.3 depicts the *else* part. Based on this traces, a language user can use an external ATP solver Z3 [42] or vanHelsing [92] and prove the *division-by-zero-free* property for the functions `y` and `x` by analyzing each TPTP trace.

We present in this chapter an improved version of the concolic execution for the

[2] See `https://casm-lang.org/syntax` for the CASM syntax description.

```
1  CASM
2
3  init test
4
5  [symbolic]
6  function x : -> Integer
7
8  [symbolic]
9  function y : -> Integer
10
11 rule test =
12 {
13     if x = 0 then
14         skip
15     else
16         y := 12 / x
17     program( self ) := undef
18 }
19 // ...
```

Listing 4.1: `Example.casm`

(open-source) CASM language and implementation. Based on the concolic execution definition by Lezuo [90], we provide two major improvements in the current presented implementation state: (1) the concolic execution generates a single TPTP trace and does not generate forked TPTP traces for each path condition (see Section 4.2); and (2) a language user only has to set ASM functions of interest to `symbolic` and each ASM function is automatically promoted to `symbolic` if there exists a path which updates that ASM function (see Section 4.3). Furthermore, we do not directly generate TPTP traces through a model-to-text transformation. We have implemented an abstraction of the TPTP model and provided an in-memory model-to-model transformation. This design decision allows us to directly (re)use in the CASM compiler the transformed TPTP instance either for further analysis, in-memory evaluation, or emitting to a textual representation in order to use an external solver.

4.2 CASM Concolic Execution and TPTP Model

CASM is a concrete ASM implementation with a strongly typed inferred specification language. The concolic execution is implemented as forward symbolic execution by reusing and extending the AST based concrete execution[3].

Due to the CASM compiler design [117], the symbolic constant, calculation, and environment handling is directly implemented on the CASM IR level[4] (see Chapter 3). Our own TPTP implementation[5] supports in-memory model-to-model transformation based on the SMT and Boolean Satisfiability Problem (SAT) solver Z3 [42] to invoke a Z3-based evaluation without external tooling.

[3] See `https://github.com/casm-lang/libcasm-fe/pull/206` for CASM front-end changes.
[4] See `https://github.com/casm-lang/libcasm-ir/pull/29` for CASM mid-end modifications.
[5] See `https://github.com/casm-lang/libtptp/pull/5` for TPTP model implementation.

```
1 tff(symbolNext, type, sym2: $int).
2 fof(id0,hypothesis,x(1,sym2)).
3 fof('Example.casm:13',hypothesis,sym2=0).
4 fof(id1,hypothesis,x(2,sym2)).
5 fof(final0,hypothesis,x(0,sym2)).
```

Listing 4.2: If-Then-Branch TPTP Trace of `Example.casm` by Lezuo [90]

```
1  tff(symbolNext, type, sym2: $int).
2  fof(id0,hypothesis,x(1,sym2)).
3  fof('Example.casm:13',hypothesis,sym2!=0).
4  tff(symbolNext, type, sym4: $int).
5  tff(symbolNext, type, sym5: $int).
6  fof(id1,hypothesis,y(1,sym5)).
7  fof(id2,hypothesis,x(2,sym2)).
8  fof(id3,hypothesis,y(2,sym4)).
9  fof(final0,hypothesis,x(0,sym2)).
10 fof(final1,hypothesis,y(0,sym4)).
```

Listing 4.3: Else-Branch TPTP Trace of `Example.casm` by Lezuo [90]

Since each ASM function can be explicitly selected to be evaluated as symbolic state (annotation syntax), a complete selection of all available ASM functions inside a specification would enable a full symbolic execution of the provided specification. So far we support all basic ASM rules in the transformation except for symbolic `iterate` rules consisting of symbolic path conditions.

Listing 4.4 depicts the same *division-by-zero-free* running example as shown in Listing 4.1 with one small change. In this listing the function `y` is not explicitly set to `symbolic`, because the function of interest we want to analyze is the function `x`. Function `y` gets implicitly set to `symbolic` through a novel compiler analysis pass (see Section 4.3) in order to provide symbolic consistency for the specified update to function `y` where function `x` is used in the division operation (see Listing 4.4 at Line 16).

Listing 4.5 corresponds to the result TPTP trace of the concolic execution. A first look at this TPTP trace gives the impression that it is longer than both TPTP traces combined of the previous implementation depicted in Listing 4.2 and Listing 4.3, but besides the path condition fork there is a huge difference in the form of the trace representation itself.

Lezuos' [90] implementation uses mixed First Order Form (FOF) and Typed First Order Form (TFF) formulae to represent the state evolving which fully complies to the deprecated TPTP versions before 7.0 [149]. Since the latest major revision 7 of TPTP the mixing of FOF and TFF does not work anymore, because variables and constants in FOF formulae are assumed to be in the same infinite domain, which is not the case for any type in a TFF formulae [149]. The later implies that each variable or constant in a TFF formulae is not equal to any variable or constant in a FOF formula. Therefore, we generate a fully typed TPTP trace by using only TFF formulae in the trace result. A transformed TPTP trace consists of four parts: (1) type declarations for intermediate calculations (see Listing 4.5 Line 1 to 7); (2)

```
1  CASM
2
3  init test
4
5  [symbolic]
6  function x : -> Integer
7
8  // concrete, not set symbolic
9  function y : -> Integer
10
11 rule test =
12 {
13     if x = 0 then
14         skip
15     else
16         y := 12 / x
17     program( self ) := undef
18 }
```

Listing 4.4: `Example.casm`

language operand definitions (see Listing 4.5 Line 8); (3) all function definitions (see Listing 4.5 Line 9 to 10); and (4) the actual trace itself (see Listing 4.5 Line 11-21). Each line in the TPTP trace represents a TFF formulae.

Since in TPTP each variable can only be used once and there exists no notion of time, each ASM function gets mapped to a TPTP predicate with 2 or more arguments where the first argument represents an Integer based time. Similar to the definition by Lezuo [90], we use time at `1` to represent the initialization of ASM functions. Time at `0` equals the termination of an ASM execution. This encoding provides an elegant way to describe start and termination constraints, since the times are known before the concolic execution starts.

Furthermore, since CASM supports block rules (parallel execution semantics) and sequential rules (sequential execution semantics) the handling of parallelism is an important issue. The evolving of function states (ASM steps) is encoded in the time value of each function in the first argument. Sequential rule computations which create *pseudo update-sets* [94] are not shown and tracked in the TPTP trace except for the remaining update to functions.

4.3 ASM Function Promotion and Symbolic Consistency

Due to the possibility that some ASM functions in a CASM specification can be marked as `symbolic`, the concolic execution can reach an interpretation of the ASM specification where a symbolic value or calculation could be used in an update rule to a concrete ASM function. This would abort the concolic execution and would lead to an execution error, because the symbolic consistency is violated. Therefore, we implemented a symbolic consistency analysis in the compiler pass pipeline which

```
1  tff(2,type,'%0':$int).
2  tff(4,type,'%1':$o).
3  tff(6,type,'%2':$int).
4  tff(10,type,'%3':$int).
5  tff(12,type,'%4':$int).
6  tff(14,type,'%5':$int).
7  tff(17,type,'%6':$int).
8  tff(7,hypothesis,'#div#i':($int*$int*$int)>$o).
9  tff(0,hypothesis,'@x':($int*$int)>$o).
10 tff(1,hypothesis,'@y':($int*$int)>$o).
11 tff(3,hypothesis,'@x'(1,'%0')).
12 tff(5,hypothesis,'%1'<=>('%0'=0)).
13 tff(8,hypothesis,~'%1'=>('#div#i'(12,'%0','%2'))).
14 tff(9,hypothesis,~'%1'=>('@y'(2,'%2'))).
15 tff(11,hypothesis,'@y'(1,'%3')).
16 tff(13,hypothesis,  ('%1' =>('%3'='%4'))&
17                    ((~'%1')=>('%2'='%4'))).
18 tff(15,hypothesis,'@x'(1,'%5')).
19 tff(16,hypothesis,'@x'(0,'%5')).
20 tff(18,hypothesis,'@y'(2,'%6')).
21 tff(19,hypothesis,'@y'(0,'%6')).
```

Listing 4.5: TPTP Trace of `Example.casm`

analyses in advance which concrete ASM functions will be updated by symbolic values. Note that updating a symbolic ASM function with a concrete value (e.g. a numeric value) is possible and does not violate symbolic consistency.

The symbolic consistency pass is an AST-based compiler analysis pass and checks if any function update produces a symbolic conflict. Each function, rule parameters, and expression AST node gets annotated by the analysis which labels the nodes either *symbolic*, *concrete*, or *unknown*.

Depending on the annotated functions through the annotation syntax, all functions are labeled either *symbolic* or *concrete* and all other nodes in the AST are labeled *unknown* at the beginning of the analysis. Since CASM supports named rule calls, each possible rule call hierarchy starting from the `init` statement has to be evaluated in order to determine symbolic consistency. The analysis derives in a step-by-step manner a Rule Call Graph (RCG) where each callable rule has to go through four states – *init*, *started*, *evaluated*, and *finished*. The resulting RCG is used to derive the final symbolic function promotion which assures symbolic consistency.

We implemented a proper reporting of ASM functions which are promoted to symbolic. Listing 4.6 depicts a console output of our CASM interpreter Command Line

```
1 casmi: info: promoting function 'y' to be symbolic, because function is
2 updated with symbolic value.
3 Example.casm:16:8..16:19
4     y := 12 / x
5     ^---------^
```

Listing 4.6: CLI Tool Information of ASM Symbolic Function Promotion

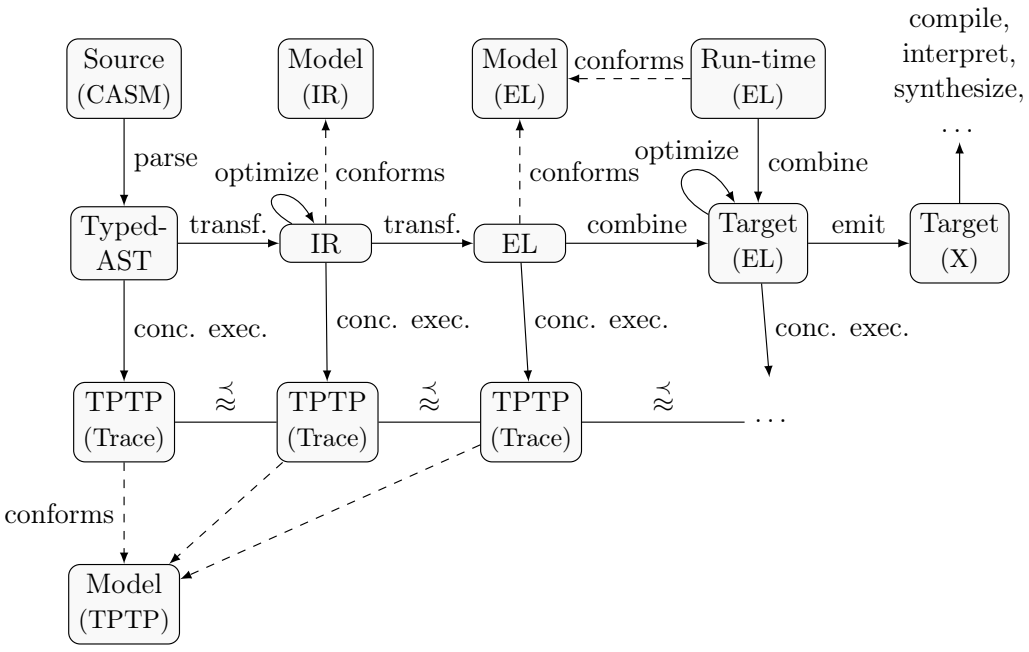


Figure 4.1: CASM Compiler Translation Validation Concept

Interface (CLI) tool named `casmi`[6] which evaluated in concolic/symbolic execution mode the `Example.casm` specification shown in Listing 4.4 and outputs an information message that function `y` gets promoted to a symbolic ASM function.

4.4 Discussion

Based on the derived concolic execution transformation described in this chapter along with the new TPTP trace generation the current effort is directed to derive a complete CASM compiler internal *translation validation* [90]. Figure 4.1 depicts the ongoing work. For every intermediate model transformation, a corresponding TPTP trace will be generated and either checked for equivalence inside the same abstraction level or between two different model levels.

4.5 Conclusion

In this chapter, we describe an improved ASM based concolic execution approach which is implemented for the CASM language and its framework.

Novel about this contribution is that the transformation of an ASM specification towards a TPTP model instance is performed through an in-memory model-to-model transformation which allows either further in-memory analysis, optimization, and evaluation of the TPTP instance or a flexible model-to-text transformation into a TPTP textual representation. Furthermore, the implemented approach only generates a single TPTP trace and promotes non-symbolic ASM functions to symbolic ones if the symbolic consistency is violated which is determined in advance through a symbolic consistency pass.

With our new concolic execution approch we aim at a complete translation validation of the CASM compiler implementation itself by checking each internal transformation step of the intermediate models [117]. Moreover, due to the introduction of state and behavioral separation in the CASM language [116], we are currently investigating the ability of automated semantic checking for imported ASM rules from loaded libraries or modules.

[6] See `https://github.com/casm-lang/casmi/pull/12` for the CLI tool `casmi`.

"Writing code is often a constant struggle against distraction." – Joy of Clojure

CHAPTER 5

Structuring Specifications

Nowadays, as in other state-based formal methods, the proposed specification languages for ASMs still lack easy-to-comprehend language constructs for type abstractions to describe reusable and maintainable specifications. Almost all built-in behaviors are implicitly defined inside a concrete ASM language implementation and thus, the behavior is hidden from the language user. In this chapter[1], we present a new ASM syntax extension based on traits, which allows the specifier (language user) to define new type abstractions in the form of structure and behavior definitions to reuse, maintain, structure, and extend the functionality in ASM specifications. We describe the proposed language construct by defining its syntax and semantics. The decision to use a trait-based syntax extension over other object-oriented language constructs like interfaces or mixins was motivated and driven by the results of previously conducted empirical studies (see Chapter 6, Chapter 7, and Chapter 8). Moreover, we outline the implementation of the trait-based syntax extension in our CASM language implementation on AST level in the new compiler design as presented in Chapter 2.

5.1 Introduction

In 1993, Gurevich [63] introduced the ASM theory, which is a well-known state-based formal method consisting of transition rules and algebraic functions. It has been used extensively by scientists for a broad research field ranging from software and hardware to system engineering perspectives in order to specify, analyze, and verify systems in a formal way. ASMs are used to formally describe the evolution of function states in a step-by-step manner[2] and are used to describe sequential, parallel, concurrent, reflective, and even quantum algorithms.

Based on the ASM theory by Gurevich [63], several theory improvements and ASM-based language implementations were developed, which were summarized by Börger and Stärk [26] and Börger and Raschke [25]. Prominent ASM languages

[1] The content of this chapter is a revised version of the ABZ'20 paper [116].
[2] ASM theory was formerly called *Evolving Algebra*.

and tools are AsmetaL [58], CASM [117], and CoreASM [50]. Today, a common thread in the various ASM languages and tools, as well as in most other state-based formal methods, is that the proposed specification languages lack easy-to-comprehend abstractions to describe reusable and maintainable type specifications. While very few have embraced basic object-oriented abstractions such as classes and inheritance, more advanced type abstractions are usually missing. Therefore, in this chapter we propose a new language construct for ASM specification languages to express type abstractions in the form of traits [38] to modularize specifications into structural state and behavioral parts.

5.2 Motivation

Modern object-oriented languages offer a variety of advanced type abstractions, and most offer either interfaces [28], mixins [53], or traits [38] in addition to classes and inheritance concepts. Interfaces establish a protocol and define method signatures to which a type has to conform [28]. They are often compared to a contract. Mixins define reusable behavior and structure that can be used to combine and form new types [53] [27]. Traits are similar to interfaces except that they can define stateless behavior which depends on the trait itself [134]. There is a heated debate in the object-oriented community[3], which of these abstractions is best suited to promote reusable and maintainable type specifications, and many implementations combine different language constructs to define type abstractions. A notable example would be the programming language Scala [111], which offers a trait syntax that is similar to the Java 8 [126] interface syntax and offers mixins type abstractions through the class-based implementation and extension syntax. Another example of mixed type abstraction concepts, namely interfaces and traits, can be found in the programming language Rust [101], where the language user has to express every interface definition through traits, and the types have to conform to specified traits and implement all required functionalities.

In the world of ASMs, only AsmL [65] has introduced an object model in the language through classes and interfaces to represent type abstractions, and to achieve structuring of the ASM specifications. Only the ASM implementation and language XASM by [3] has introduced a sub-ASM construct to achieve a component-based modularization approach. A more generic concept called *ambient* ASMs [25] introduces the possibility to achieve hierarchical state partitioning through nesting of context-sensitive (sub)program environments. Based on this state of the art, we started to investigate the introduction of a new type abstraction language construct in ASMs. But which language construct is suitable for ASMs to represent such type abstractions?

Basically every language construct for forming type abstractions is suitable for ASMs, but it influences the understandability of the language considerably. For such

[3] See e.g. `https://stackoverflow.com/questions/925609` for the discussion page.

an ASM extension, we consider the following properties important: (1) reuse and embed existing specifications; (2) describe built-in behavior of a language itself in the language; and (3) allow encapsulation of ASM states and corresponding behavior through modularization.

Driven by the properties and questions raised, we conducted three empirical studies to determine, which language construct – interfaces, mixins, or traits – is most understandable to ASM language users for expressing type abstractions (see Chapter 6, Chapter 7, and Chapter 8). The result of the experiments showed that the participants with strong object-oriented backgrounds (highly familiar with interfaces, not familiar with traits at all) had a similar to equal understanding of an interface and traits language construct in the experimental ASM syntax variants. Mixins, on the other hand, had a significantly lower understandability compared to traits and interfaces. Since the interface and traits type abstraction language constructs offer a similar to equal understandability, and novice language users seem to understand traits without even knowing the concept of traits, we investigated introducing traits into ASMs.

Moreover, the object-oriented communities often discuss traits more favorably than interfaces[4] and even point out that *"Traits are Interfaces"*[5] just with code-level reuse functionality. To gain a better understanding of how specifiers (language users) comprehend such trait-based specifications, we performed an eye-tracking experiment [141], where we observed the participants' gaze patterns (see Chapter 8). The results of this experiment showed that the participants could easily distinguish between behavioral and non-behavioral aspects of a given specification, when we applied our trait-based language construct to form state/behavior type abstractions.

5.3 Traits Fundamentals

In 1982, Curry et al. [38] introduced traits for the first time for the Self programming language. Roughly speaking, traits can be seen as an inter-object communication between interfaces and mixins.

Interfaces [28] only have the capability to define a finite set of operations. Every operation (declaration) has a name and a typed relation. Each class (structure) that wants to use the interface has to implement the operation (definition). Each implemented interface operation can only access the state information of the corresponding class (structure). Therefore, redundant behavior has to be specified in multiple classes (structures) and cannot be fostered at the same location with proper reuse functionality.

Apart from the definition of behavior (declaration and/or definition of operations), mixins [53] allow defining state information as well. Moreover, mixins enforce the *"dependency inversion principle"* (pattern) [100], which states that higher level modules

[4]See e.g. `https://stackoverflow.com/questions/9205083` for the discussion page.
[5]See e.g. `https://blog.rust-lang.org/2015/05/11/traits` for traits in Rust elaboration.

shall be decoupled from lower level modules through abstractions. The problem here are the linear composition capabilities of mixins due to the included state information of objects inside the mixin.

Traits are placed between interfaces and mixins because they offer the definition of operation declarations and operation definitions, whereas the behavior can only depend on the trait itself. Moreover, a trait does not contain any state information (stateless). Based on these capabilities, traits offer a variety of advantages compared to interfaces and mixins, which are summarized by Schärli et al. [134].

The biggest advantage is that traits avoid the *"diamond problem"* [99] that occurs in multi-inheritance object-models because it does not require linear composition. If a class (structure) implemented multi-inherited behavior, the conflict (ambiguous behavior) would have to be manually resolved (explicit conflict resolution) by the language user.

Fisher and Reppy [52] defined properties, operations, a typed calculus, and proven type soundness for traits to achieve statically-typed traits. A trait (denoted as upper case letters) consists of one or multiple unique operations (denoted as lower case letters), so a trait equals a set of operations (i.e. $A = \{a_1, \ldots, a_n\}$).

Operations over traits are defined as: (1) *symmetric sum*: the addition of two traits results in a new trait (e.g. $A + B = X$); (2) *asymmetric sum*: the addition of operations with possible overwrites to an existing trait results in a new trait (e.g. $A + b = C$); (3) *alias*: the addition of a new unique name for an existing operation in a trait results in a new trait (e.g. $(A \setminus \{a_1\}) + b = D$); and (4) *exclusion*: removing an operation from a trait results in a new trait (e.g. $A \setminus \{a_1\} = E$).

The properties of traits are: (1) trait composition is commutative and thus, the order of implementing a trait does not affect the behavior of a trait (e.g. $X = A + B \wedge Y = B + A \Rightarrow X = Y$); and (2) nested traits hierarchies are equal to their flatted representation and thus, the trait hierarchy does not affect the behavior of a trait (e.g. $X = A + Z \wedge Z = B + C \Rightarrow X = A + B + C$).

5.4 A Trait-Based Construct for ASMs

This section proposes our trait-based language construct to extend the syntax of ASM specification languages.

The syntax rules are defined and expressed in Backus–Naur Form (BNF) (see Listing 5.1). The semantics of the proposed trait-based syntax extension is defined by lowering and transforming the new syntax elements to appropriate Turbo ASM [26] equivalent definitions (see example trait-based ASM Listing 5.2 and the transformed Turbo ASM Listing 5.3). The ASM specifications presented use the syntax of the CASM specification language[6]. The trait-based syntax extension is divided into three parts, namely *structural types*, *basic type behavior*, and *extended type behavior*.

[6]For the CASM syntax description, see: `https://casm-lang.org/syntax`

Structural Types

In order to modularize the states (functions not classified as derived) in ASM, we introduce a *structural type* construct (see Listing 5.1, Line 1-4), which allows a language user to group one or multiple functions together (similar to members of an object-oriented class) to form a new *structure* type (see `StructureDefinition` grammar rule).

```
1  // Structural Types
2  StructureDefinition ::= 'structure' Identifier '=' '{' ( FunctionDefinition )+ '}'.
3  StructureLiteral    ::= [Type] '{' [Identifier ':' Term (',' Identifier ':' Term)*] '}'.
4  Literal             ::= StructureLiteral | /* other literals */.
5  // Basic Type Behavior
6  ImplementDefinition     ::= 'implement' Identifier '=' '{'
7                              ( ObjectRuleDefinition | ObjectDerivedDefinition )+ '}'.
8  ObjectRuleDefinition    ::= 'rule' Identifier '(' 'this'
9                              ( ',' Identifier ':' Type )* ')' [ '->' Type ] '=' Rule.
10 ObjectDerivedDefinition ::= 'derived' Identifier '(' 'this'
11                             ( ',' Identifier ':' Type )* ')' '->' Type '=' Term.
12 MethodCall              ::= Term '.' Identifier ['(' Term (',' Term)* ')'].
13 CallRule                ::= MethodCall | ( Identifier [ '(' Term (',' Term)* ')' ] ).
14 Term                    ::= MethodCall | 'this'
15 // Extended Type Behavior
16 BehaviorDefinition       ::= 'behavior' Identifier '=' '{'
17                              ( ObjectRuleDeclaration | ObjectDerivedDeclaration
18                              | ObjectRuleDefinition  | ObjectDerivedDefinition )+ '}'.
19 ImplementForDefinition   ::= 'implement' Identifier 'for' Identifier '=' '{'
20                              ( ObjectRuleDefinition  | ObjectDerivedDefinition )+ '}'.
21 ObjectRuleDeclaration    ::= 'rule'    Identifier ':' 'Object' ('*' Type)* '->' Type.
22 ObjectDerivedDeclaration ::= 'derived' Identifier ':' 'Object' ('*' Type)* '->' Type.
```

Listing 5.1: Trait-Based ASM Syntax Extension

```
1  structure X = {
2    function f1 : -> Integer
3    function f2 : Integer -> Boolean
4  }
5
6
7
8
9
10
11 rule R1 =
12   let v1 = X{ f1: 1,
13     f2: (2) -> false } in skip
14 implement X = {
15   derived d1(this) -> Integer =
16     this.f1
17
18   rule R2( this, a1 : Integer ) =
19     if a1 > -5 and this.d1 < 5 then
20       this.f2(a1) := true
21 }
22 behavior Y = {
23   derived d2 : Object -> Integer
24
25   derived d3( this ) -> Boolean
26     = this.d2 * this.d2 > 100
27 }
28 implement Y for X = {
29   derived d2(this) -> Integer =
30     this.f1
31 }
32 // ...
```

Listing 5.2: Trait-Based ASM

```
1  domain X
2  function X_f1 : X -> Integer
3  function X_f2 : X * Integer -> Boolean
4  rule X_instantiate( a1 : Integer
5  , a2 : Integer -> Boolean ) -> X =
6    let object = new X in {
7      X_f1( object ) := a1
8      X_f2( object ) := a2
9      result := object
10   }
11 rule R1 =
12   let v1 = X_instantiate( 1,
13     { (2) -> false } ) in skip
14
15 derived X_d1( this : X ) -> Integer =
16   X_f1(this)
17
18 rule X_R2( this : X, a1 : Integer ) =
19     if a1 > -5 and X_d1(this) < 5 then
20       X_f2(this, a1) := true
21
22
23
24
25 derived X_d3( this : X ) -> Boolean
26     = X_d2(this) * X_d2(this) > 100
27 }
28
29 derived X_d2(this:X) -> Integer =
30     X_f1(this)
31
32 // ...
```

Listing 5.3: Turbo ASM Equivalent

Each structure type defines a trait type through the defined state functions. The access to these functions is only allowed inside a proper basic behavior definition to clearly specify the access to an instantiated structure's state over dedicated behaviors (data encapsulation).

We extend the mathematical notation (introduced in Section 1.6): (1) $\mathcal{S}$ defines a finite set of all structure type names; (2) M defines a unary relation of structure names to a finite set of function names ($M : \mathcal{S} \rightarrow \{f_1, \ldots, f_n\} \ \wedge \ n \ \in \mathbb{N}_{>0}$) [7]; (3) $\mathcal{B}$ defines a finite set of all behavior (trait) type names, whereas T represents a behavior (trait) type name ($T \in \mathcal{B}$); and (4) B defines a unary relation of behavior type names to a finite set of corresponding behavior type names and thus, the relation B represents the behavior (trait) type hierarchy ($B : \mathcal{B} \rightarrow \{T_1, \ldots, T_n\} \wedge n \ \in \mathbb{N}_{>0}$). (5) I defines a unary relation of behavior type names to a finite set of corresponding function f, derived function d, and rule R names ($I : \mathcal{I} \rightarrow \{f_1, \ldots, f_n, d_1, \ldots, d_m, R_1, \ldots, R_w\} \ \wedge \ n, m, w \ \in \mathbb{N}_{>0}$).

A `StructureDefinition` defines a new unique *structure* type name X_s, which gets added to $\mathcal{S}$, and M contains the corresponding mapping to the n nested functions $\{f_1, \ldots, f_n\}$ ($\Rightarrow X_s \in \mathcal{S} \ \wedge \ M(X_s) = \{f_1, \ldots, f_n\} \ \wedge \ M(X_s) \cap \Sigma = \emptyset$).

The `Literal` grammar rule gets extended by a `StructureLiteral` to define a term t consisting of a non-negative number n assignments. Each assignment binds a term t_n of position n to a unique variable name v_n resulting in $t = \{v_1 = t_1, \ldots, v_n = t_n\} \ \wedge \ n \in \mathbb{N}_{\geq 0}$. The term t initializes the structure S.

Lowering The transformation to an equivalent Turbo ASM is defined as:

(1) $\forall X_s \in \mathcal{S}, \ \forall f \in M(X_s)$ expand the argument types $\{X_1, \ldots, X_n\}$ of function f of non-negative arity n with the structure type X_s resulting in a new function f' of non-negative arity $n + 1$ with argument types $\{X_s, X_1, \ldots, X_n\}$, add f' to Σ, add f' to $I(X_s)$, and remove f from $M(X_s)$.

(2) $\forall X_s \in \mathcal{S}$ create a named rule $X_{instantiate}$, which has n arguments $\{a_1, \ldots, a_n\}$, and a target type of X_s, add $X_{instantiate}$ to $\mathcal{R}$, add X_s to $\mathcal{U}$, add X_s to $B(X_s)$, and remove X_s from $\mathcal{S}$. The rule $X_{instantiate}$ creates a new symbol *object* of X_s and performs an update to the corresponding function f'_n for all arguments a_n and returns the value of the *object*. Therefore, the `StructureLiteral` gets directly mapped in a transformation to the $X_{instantiate}$ rule. After the transformation, all structures shall be consumed ($\mathcal{S} = \emptyset$).

An example of a defined structural type and its initialization is depicted in Listing 5.2. At Line 1, we defined the structure `X` consisting of two functions `f1` and `f2`. At Line 12, we can see a rule `R1`, which uses the structure literal syntax to create an object of type `X`. Listing 5.3 shows the corresponding Turbo ASM representation. We can see that the structure definition results in a domain definition `X` along with two function definitions `X_f1` and `X_f2` with an additional preceded type argument of `X`

[7] $\mathbb{N}_{>0} = \{n \in \mathbb{N} \mid n > 0\}$

(Lines 1 to 3). Moreover, we can observe the created `X_instantiate` rule in Line 5 and its usage in Line 13.

A behavior has to be defined in order to modify (access) the state of an instantiated structure. We distinguish between basic and extended type behavior.

Basic Type Behavior

A *basic type behavior* (see Listing 5.1, Line 5-14) defines a set of rules and derived functions, which are associated with a certain domain type. We introduce a new `ImplementDefinition` to define a basic behavior consisting of one or more object-based derived function and/or rule definitions.

The syntax for `ObjectRuleDefinition` and `ObjectDerivedDefinition` introduce a new keyword `this` as the first argument for all object-based rule and/or derived function definitions. The type of the argument variable `this` equals the type of the `ImplementDefinition` and it enables the access to the domain's or structure's behavior.

The access happens through a `MethodCall` syntax, which uses a dot operator between a term, a target name, and a non-negative arity of arguments. The target name can be a function name or a rule name.

Lowering The transformation to an equivalent Turbo ASM is defined as:

(1) $\forall X \in \mathcal{B}, \forall d, R \in I(X)$ create a d' of d and R' of R, set in d' and R' the argument type for argument `this` to X, add d' to Σ, and add R' to $\mathcal{R}$.

(2) each `MethodCall` with $t \,.\, u(a_1, \ldots, a_n)$ gets transformed to $u'(t, a_1, \ldots, a_n)$, where u' is either a transformed function f', derived function d', or rule R' name.

Listing 5.2 defines a basic behavior (Line 14) for type X (example structural type from Listing 5.2) consisting of a derived function `d1` (Line 15) and a rule `R2` (Line 18). The transformed Turbo ASM representation is shown in Listing 5.3. In this example, we can observe that all method calls are resolved to the corresponding function names `X_f1`, `X_f2` and the derived name `X_d1` of structural type `X`.

Extended Type Behavior

An *extended type behavior* (see Listing 5.1, Line 15-22) defines a set of rules and derived functions, and forms a new type in the type system. If a domain and/or structural type wants to use the functionality, it has to implement the extended behavior.

The `BehaviorDefinition` defines an explicit trait with type name consisting of zero or more `ObjectRuleDeclaration` rule names and/or `ObjectDerivedDeclaration` derived function names. Please note that for all object-based declarations we introduced a generic `Object` argument type at the first position. The `Object` type gets checked against the domain or structural type which is implementing this declared behavior. A specifier can use the `Object` type for any other argument or target type in a

declaration. Additionally, a trait can define a default behavior through zero or more `ObjectRuleDefinition` rule names and/or `ObjectDerivedDefinition` derived function names, which depends only on the functionality of the trait itself. Each domain and/or structural type that wants to support a certain behavior has to specify an `ImplementForDefinition` and provide the missing definitions of the trait declarations. If the trait defines a default behavior, the domain and/or structural type inherits this definition. This enables code reuse capabilities.

Lowering The transformation to an equivalent Turbo ASM is equivalent to the defined basic behavior transformation (lowering) from Section 5.4.

Listing 5.2 defines a behavior Y (Line 22) consisting of two derived functions `d2` and `d3` (Lines 23 and 25). Moreover, on Line 28 the behavior Y gets implemented for type X by specifying the missing behavior of `d2`. The transformed Turbo ASM representation is shown in Listing 5.3. It contains the transformed derived function `X_d2` of the implementation and the defined behavior of `Y` as `X_d3`.

Based on the proposed trait-based syntax, it is possible to express the operator behaviors and other language features through traits directly in the ASM language itself. Listing 5.9 shows a behavior definition of the `Equality` trait (see Section 5.4) which declares the operation `equal` and defines and operation `unequal` by using the negated result of the `equal` behavior. This behavior is needed by the evaluation of the equal operator (`=`) in an ASM language. Moreover, if a function uses a domain or structure type as an argument type, an implemented `Equality` behavior is needed to compare location values during the evaluation. Listing 5.10 defines a function `f3` (Line 1) with the structure `X` as argument. In order to have a valid specification, the language user has to define the semantics of the `Equality` behavior for the structure `X` as well (Lines 3 to 7). The transformed Turbo ASM representation is shown in Listing 5.11.

Example Specification

Listing 5.2 depicts an example trait-based ASM specification using all new syntax grammar rules and Listing 5.3 depicts the equivalent semantics-preserving Turbo ASM specification. The proposed trait-based syntax extension is realized in our

```
1 function f4 : -> X = {
2   f1: 1,
3   f2: { ( 2 ) -> false }
4 }
5
6 // ...
```

Listing 5.4: Trait-Based ASM

```
1 function f4 : -> X
2 rule f4_initially = {
3   f4 := X_instantiate( 1,
4     { ( 2 ) -> false } )
5 }
6 // ...
```

Listing 5.5: Turbo ASM

CASM language implementation[8]. In order to provide a clean solution of the defined transformations of the trait-based ASM to an equivalent Turbo ASM, we updated our CASM language front-end implementation and introduced two new internal AST representations before the specification gets transformed to the CASM-IR [117] which is introduced in Chapter 3.

Prelude Specification

By introducing the proposed trait-based construct, we were able to explicitly specify the behavior of the CASM language itself in CASM in the form of a *prelude*[9] specification, which gets automatically loaded (imported) for every parsed CASM specification.

Each functionality of the CASM language (e.g. operators) is mapped to a behavior (trait) in the prelude specification. The language user can explore and extend the

[8] See `https://github.com/casm-lang/libcasm-fe/pull/205` for sources.

[9] See `https://github.com/casm-lang/libcasm-fe/blob/master/lib/CASM.casm` for the prelude specification.

```
1  CASM init test
2  behavior Incrementing = {
3    rule increment : Object -> Void
4    rule doubleIncrement(this) = {
5      increment increment
6    }
7  }
8  structure Counter = {
9    function value : -> Integer
10 }
11 implement Counter = {
12   derived currentValue(this) -> Integer
13     = this.value
14 }
15 implement Incrementing for Counter = {
16   rule increment(this) =
17     this.value := this.value + 1
18 }
19 function a : -> Counter = { value: 1 }
20 function b : -> Counter = { value: 0 }
21 rule test = {|
22   a.increment
23   b.doubleIncrement
24   if a = b then
25     println( "Equal" )
26   else
27     println( "Unequal" )
28 |}
```

Listing 5.6: Counter Specification (counter.casm)

```
error: counter.casm:24:6..24:11: invalid binary operator '=',
because the behavior 'Equality' is not defined for type 'Counter'
  if a = b then
     ^~~~^
```

Listing 5.7: Type Inference Error Message

behaviors of CASM in CASM. Moreover, the prelude specification reduced the complexity of the CASM implementation.

5.5 Implementation

The parsed CASM specification gets transformed to a Concrete Syntax Tree (CST) representation. The CST gets transformed to the AST representation, which includes syntactic sugar rewrites and the widening of the functions argument relation of structural types. as specified in the lowering of Section 5.4. Moreover, the AST gets transformed to the Lowered Abstract Syntax Tree (LST) representation. The latter only defines AST nodes supported by an equivalent Turbo ASM. The AST to CST transformation implements the lowering as defined in Section 5.4.

Lets have a look at a small example specification to demonstrate the trait-based syntax extension. Listing 5.6 shows a specification which defines an `Incrementing` behavior (Lines 2 to 7). Moreover, a structural type `Counter` is defined (Lines 8 to 10) containing a nullary function `value` of target domain `Integer`. A basic behavior for `Counter` is provided to access the object's state (Lines 11 to 14). The `Counter` behavior gets extended by implementing the `Incrementing` behavior (Lines 15 to

```
29 implement Equality for Counter = {
30   derived equal( this, other : Counter ) -> Boolean =
31     (this.value = other.currentValue)
32 }
```

Listing 5.8: Extension of Counter Specification (counter.casm)

```
1 behavior Equality = {
2   derived equal : Object * Object -> Boolean
3   derived unequal( this, other : Equality ) -> Boolean =
4     not this.equal( other )
5 }
```

Listing 5.9: Prelude `Equality` Behavior Specification (*E*xcerpt of `CASM.casm`)

```
1 function f3 : X -> Integer
2
3 implement Equality for X = {
4   derived equal
5   (this, other : X) -> Boolean =
6     this.f1 = other.f1
7 }
8
9
10
11 // ...
```

Listing 5.10: Trait-Based ASM

```
1 function f3 : X -> Integer
2
3
4 derived X_equal
5 (this : X, other : X) -> Boolean =
6   X_f1(this) = X_f1(other)
7
8 derived X_unequal
9 (this : X, other : X) -> Boolean =
10   Boolean_not(X_equal(this,other))
11 // ...
```

Listing 5.11: Turbo ASM

18). Two functions are defined with target type `Counter` (Lines 19 to 20). A single execution agent (Line 1) evaluates the rule `test`, increments the function value `a`, increments the function value `b` twice (Lines 23 to 24), and checks if the contents of `a` and `b` are equal (Line 24). Since the behavior of the equal operator at Line 24 is not yet defined for the structural type `Counter`, the error message shown in Listing 5.7 appears during a type inference analysis of the specification. To provide a correct counter specification, it has to be extended with the implementation of the `Equality` behavior for the structure `Counter` as shown in Listing 5.8.

5.6 Related Work

To the best of our knowledge, traits have so far not been introduced in an ASM language. Thus, we compare CASM against the current state of the art of ASM languages to find out if and how they support certain language features to achieve structuring and modularization of state and behavior in specifications. We extend the comparison to the scope of the ABZ conference formal methods as well as to some selected object-oriented programming languages.

Thus, the comparison analyzes if and how the following language features are realized: (1) state abstraction; (2) behavior abstraction; (3) composition capabilities; (4) inheritance model; (5) modularization; (6) namespace concept; (7) syntax extension capabilities; and (8) semantics extension capabilities. Table 5.1 provides a complete overview of the following language features discussed.

In contrast to CASM, AsmL [65] is the only other ASM language to define state and behavior abstractions, which is achieved by encapsulating the state and behavior in classes, since this language is directly integrated into the .NET run-time. Where AsmL uses single inheritance to allow sub-typing, CASM only relies on behavior (trait) types and the accompanying composition capabilities. Both provide the ability of a namespace concept. XASM [3] introduced a composition concept as well, but in the form of a component-based approach to form sub-ASM hierarchies. The composition capabilities are limited compared to CASM since no separation of state and behavior abstraction is possible. Moreover, XASM sub-components (modules) cannot be reused in other specifications because the connection has to be bidirectionally defined by the sub-ASMs. AsmetaL [58] and CoreASM [50] do not provide any for state or behavior abstraction concepts. Both offer a light-weight module concept which is based on a simple file-include mechanism. In comparison, CASM offers a full module-based import and namespace resolving of specification symbols. Besides CoreASM, none of the other ASM implementations offer the extension of syntax and semantics, but the extensions have to be written in the form of Java plugins. With the trait-based construct, CASM enables the extension and definition of custom semantics directly in the ASM language itself.

In the scope of the ABZ, Alloy [74] has a built-in object-orientation concept by

Language	State Abstract.	Behavior Abstract.	Compo-sition	Inheri-tance
AsmetaL	-	-	-	-
AsmL	class	interface	yes	single
CASM	structure	trait	yes	-
CoreASM	-	-	-	-
XASM	-	-	yes	-
Alloy	signature	signature	yes	single
Event-B	record	-	-	-
TLA+	record	-	-	-
VDM++	class	mixin	-	multiple
Z++	class	mixin	-	multiple
C++	class	mixin	yes	multiple
Java	class	interface	yes	single
Rust	structure	trait	yes	-
Swift	class	protocol	yes	single

Language	Modulari-zation	Name-spaces	Syntax Ext.	Semantics Ext.
AsmetaL	light	-	-	-
AsmL	yes	yes	-	-
CASM	yes	yes	-	yes
CoreASM	light	-	plugin	plugin
XASM	yes	-	-	-
Alloy	yes	yes	-	-
Event-B	-	-	-	-
TLA+	yes	-	-	-
VDM++	yes	yes	-	-
Z++	-	-	-	-
C++	light	yes	operat.	operat.
Java	yes	yes	-	-
Rust	yes	yes	plugin	yes
Swift	yes	-	operat.	operat.

Table 5.1: Feature Comparison of Specification and Programming Languages

extending signatures by signatures. Alloy implements a single inheritance concept, package-based namespace and module concept. Thus, it offers composition, state, and behavior abstraction over one syntactical element. Event-B [47] and TLA+ [87], on the other hand, just offer to structure the state through records (named tuples). VDM++ [45] and Z++ [88] are object-oriented extensions of their underlying notation (VDM and Z), where both provide the ability to create state and behavior abstractions over classes with mixin capabilities through a multiple inheritance model. All existing methods in the ABZ scope besides CASM have either no state and behavior abstraction construct or one based on classes over a dedicated inheritance model.

Some well-known programming languages like C++ [148], Java [126], and Swift [57] all use classes as state abstractions. They differ in the usage of their main behavior abstraction, where C++ uses mixins (abstract, virtual, and pure virtual classes), Java uses interfaces, and Swift uses traits (protocols with extensions). Furthermore, C++ uses multiple inheritance, whereas Java uses a single inheritance model. Java does not offer syntax and semantics extension capabilities over the language operators,

which is possible in C++ through operator overloading. Since C++ and Java use an inheritance model, we have a look at another object-oriented programming language. Rust [101] offers the same state and behavior abstractions and composition capabilities as CASM and it does not have an inheritance model either. Moreover, Rust allows to extend the syntax through handwritten Rust plugins, which is not possible in CASM so far, but planned. Our trait-based syntax was influenced by Rust.

5.7 Conclusion

In this chapter, we present a trait-based construct for ASM languages. It allows to specify composable models through the usage of domain and structural type objects, where the behavior can be defined and implemented in a reusable manner. The modularization and composing of object-oriented models is achieved by specifying structural states along with their behaviors clearly separated through traits. Novel about this contribution is that ASM language users can directly define the semantics of operations over domain (structure) types through this trait-based construct in the ASM language itself. To clearly separate structure and behavior, we only allow the definition of modifications to structural objects through a proper behavior definition. Based on previously conducted empirical studies, the current state of the art, and our current proposed trait-based construct, we believe that this is the first step towards clearer and more understandable ASM specifications by separating the structural (state) and behavioral elements through dedicated definitions.

"Simple languages produce complicated solutions because you have to be verbose." – Larry Wall

CHAPTER 6

Understandability Study

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. In this chapter[1], 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.

6.1 Introduction

Several scientists of different research fields used and applied the ASM theory and its ASM method [26]. This usage ranges from software, hardware and system engineering perspectives to specify, analyze, verify, and validate systems in a formal way [129] [128]. The diversity of ASM-based applications ranges from formal specification of semantics of programming languages, such as those for Java by Stärk et al. [146] or VHDL by Sasaki [133], compiler back-end verification by Lezuo [90], software run-time verification by Barnett and Schulte [11], software and hardware architecture modeling

[1]The content of this chapter is a revised version of the JSS'21 article [119].

e.g. of UPnP by Glässer and Veanes [60], to even RISC designs by Huggins and Campenhout [72].

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 [26]. 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 [63]. Additionally, several theory improvements were provided to increase the expressiveness of ASM languages which were summarized by Börger and Stärk [26] and Börger and Raschke [25].

The landscape of developed ASM languages and the corresponding tools is rather limited nowadays. Best known are the ASM implementations AsmetaL [58] and CoreASM [50]. AsmetaL provides a feature-rich tool set to model, analyze, interpret, and generate code of described ASM specifications[3]. The core of AsmetaL is implemented and based on the EMF and provides therefore a Java-based interpreter. CoreASM[4] 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 [50]. 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 [65] and CASM [91]. 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. [91] established compilation techniques to outperform CoreASM and AsmL in terms of ASM execution performance. There are several other ASM language tool implementations like AsmGofer [136] or XASM [3], 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 [74], Event-B [2], TLA [86], VDM [19], and Z [125].

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 [103]. While very few have embraced basic object-oriented abstractions such as classes and inheritance, more advanced language constructs are usually missing. Mernik et al. [102] point out that the lack of such object-oriented abstractions in formal methods

[2]The ASM theory was formerly called *Evolving Algebra*.
[3]See `https://asmeta.github.io` for the AsmetaL project site.
[4]See `https://github.com/CoreASM` for the CoreASM project site.

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 [24] 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 [28], mixins [53], or traits [134] in addition to classes and inheritance. All of those three language construct have similar and some different properties and characteristics, which are depicted in Figure 6.1 and described as follows:

Interfaces define (typed) operations (signatures) to which an implementer of a certain interface (type) has to conform [28]. Therefore, an interface defines a so called *contract* [105]. 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 [53] [27]. 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 [134]. 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 [111], which offers a trait syntax that is similar to the Java 8 [126] 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 [101], 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

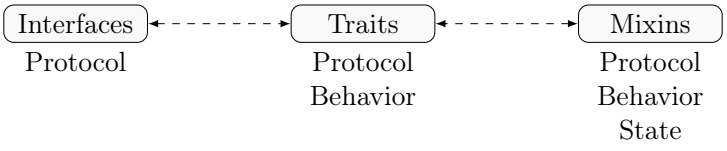


Figure 6.1: Overview of Language Construct Properties

engineers when making decisions on choosing language constructs in specification language designs and implementations.

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 [91] [94] [123] [117] 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 [152] 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[5].

Our hypotheses are influenced by the debate in the object-oriented community, which recently discuss traits often more favorably than mixins[6]. 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[7] or point out that *"Traits are Interfaces"*[8] 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

[5]See `https://cs.univie.ac.at` for faculty website.
[6]See, e.g. `https://stackoverflow.com/questions/925609`.
[7]See, e.g. `https://stackoverflow.com/questions/9205083`.
[8]See, e.g. `https://blog.rust-lang.org/2015/05/11/traits`.

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.

Structure of this Chapter

In Section 6.2, we describe ASMs, the used language and constructs used in this study, and present related studies. Section 6.3 elaborates the planning of the language construct study. In Section 6.4, we describe the execution of the experiment, while the results are presented in Section 6.5 and discussed in Section 6.6. Last but not least, we conclude the chapter in Section 6.7.

6.2 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.

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 [66], 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 [26].

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 [25] 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 6.2).

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 [64], 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* [25]. A *language user* can model transactions through *named rules* (see Section 6.2).

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 [25].

ASM Language Representative

In this study, we use the basic syntax elements from the CASM language[9] [117]. 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 6.1 illustrates the concrete syntax and some examples.

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

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 6.2 illustrates the concrete syntax and some examples which use state information from Listing 6.1.

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 [25]. Listing 6.3 illustrates the concrete syntax and an example which depends on some definitions from Listing 6.1 and Listing 6.2.

Experiment Language Construct Representations

Besides a class concept used in AsmL [65], 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,

```
1 function counter : -> Integer // variable
2
3 function personsAge : String -> Integer // hash-map
```

Listing 6.1: *Function* Definition Example

```
1 derived nextCounter -> Integer = counter + 1
2
3 derived isFullAged( name : String ) -> Boolean =
4   ( personsAge( name ) >= 18 )
```

Listing 6.2: *Derived* Definition Example

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

Listing 6.3: Named *Rule* Definition Example

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 [97] [28] 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 7.4 depicts an example specification with the *Interface* language construct[10].

[10]See `form_ifaces.pdf` at [120].

```
1  feature Formatting = {
2    derived toString : -> String
3  }
4
5  structure Person implement Formatting = {
6    function name : -> String
7    function age : -> Integer
8
9    derived getName -> String  = this.name
10   derived getAge -> Integer = this.age
11
12   rule setName( name : String  ) = this.name := name
13   rule setAge( age : Integer ) = this.age := age
14
15   // encapusalted feature implementation
16   derived toString -> String = this.getName() + (this.getAge() as String)
17 }
```

Listing 6.4: *Interfaces*-Based Example Specification

This syntax is primarily influenced by the Java programming language [126] 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 [106] [53]. To indicate that a `structure` shall provide the behavior of a `feature`, the `implement` keyword is used to extend the current `structure` implementation by

```
1  feature Formatting = {
2    derived toString -> String
3  }
4
5  implement Formatting = {
6    derived toString -> String = ""
7  }
8
9  structure Person implement Formatting = {
10   function name : -> String
11   function age  : -> Integer
12
13   derived getName -> String  = this.name
14   derived getAge  -> Integer = this.age
15
16   rule setName( name : String  ) = this.name := name
17   rule setAge( age : Integer ) = this.age := age
18
19   // overwrite of feature implementation
20   derived toString -> String = this.getName() + (this.getAge() as String)
21 }
```

Listing 6.5: *Mixins*-Based Example Specification

```
1  feature Formatting = {
2    derived toString -> String
3  }
4
5  structure Person = {
6    function name : -> String
7    function age  : -> Integer
8  }
9
10 implement Person = {
11   derived getName -> String = this.name
12   derived getAge -> Integer = this.age
13
14   rule setName( name : String  ) = this.name := name
15   rule setAge( age : Integer ) = this.age  := age
16 }
17
18 // decoupled feature implementation
19 implement Formatting for Person = {
20   derived toString -> String = this.getName() + (this.getAge() as String)
21 }
```

Listing 6.6: *Traits*-Based Example Specification

the default implementation and function state. Every default implementation can be overwritten by an explicit concrete implementation of a certain operation. Listing 6.5 depicts an example specification with the *Mixins* language construct[11]. This syntax is primarily influenced by the Scala programming language [111] 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 7.5 depicts an example specification with the *Traits* language construct[12]. This `feature` and `implement` syntax is influenced by the Rust programming language [101] trait syntax[13].

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 [108]. Apel and Batory present a case study comparing aspect and feature abstractions using a mixin layer approach to unify the two [4]. Batory et al. present another case study on achieving extensibility through product-lines and domain-specific languages using a mixin-based approach [12]. 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 [33] [41] [31] and architecture description languages that offer formal representations [112] [59]. 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 [145]. Czepa and Zdun [40] and Czepa et al. [39] have studied the

[11] See `form_mixins.pdf` at [120].
[12] See `form_traits.pdf` at [120].
[13] See `https://doc.rust-lang.org/rust-by-example/trait.html` for the discussion.

understandability of formal methods for temporal property specification using similar research methods as used in this study.

Ferrarotti et al. [51] 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 [51] concept in order to represent the abstract type (interfaces) and sub-classing mechanisms.

Related to this study, we conducted another controlled experiment [121] 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 [121] 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.

6.3 Experiment Planning

This study is structured following the guidelines by Jedlitschka et al. [77] on how empirical research shall be conducted and reported in software engineering. Moreover, the guidelines by Kitchenham et al. [82], Wohlin et al. [158], and Juristo and Moreno [80] 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. [81].

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.

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[14], 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[15], 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.

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 DSL [54] and MDD [17] 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 6.3).

In total, there were 105 participants, which were randomly allocated to the treatments (i.e. the three language construct representations in an ASM specification

[14]See `https://ufind.univie.ac.at/en/course.html?lv=053020&semester=2018S` for ASE.
[15]See `https://ufind.univie.ac.at/en/course.html?lv=052500&semester=2018S` for DSE.

language, see Section 6.2). 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. [82] using students *"is not a major issue as long as you are interested in evaluating the use of a technique by novice or nonexpert software engineers. Students are the next generation of software professionals and, so, are relatively close to the population of interest"*. This is directly reflected in this study because some of the students who participated in the experiment show several years of programming experience as well as several years of work experience in the software and/or hardware industry (see Figure 6.2c, which summarizes the participants' industrial work experiences).

Other studies by Svahnberg et al. [151] or Salman et al. [132] would argue even further and state that under certain circumstances, students are valid representatives for professionals in empirical software engineering experiments.

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 [25].

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[16] explaining the ASM language syntax and semantics **without the experiments' language construct** syntax and semantics extensions.

(2) Survey Form Three experiment survey forms[17] per experimental group and language construct contain the actual survey along **with the explicit experi-**

[16] See `info.pdf` at [120].
[17] See `form_ifaces.pdf`, `form_mixins.pdf`, and `form_traits.pdf` at [120].

ments' 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.

Variables

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 6.3) excluding breaks.

These two dependent variables are commonly used to measure the construct **understandability** (cf. Hoisl et al. [71], Czepa and Zdun [40]). The independent

variables (factors) have three treatments, namely the three different representations of language constructs *Interfaces*, *Mixins*, and *Traits*.

Hypotheses

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 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*.

6.4 Experiment Execution

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

Preparation

Two weeks before the experiment we handed out the preparation material (the experiment *information sheet*, see Section 6.3) through an e-learning platform[18].

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

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 6.2).

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 6.4. At the beginning of the experiment, every participant received a random experiment *survey form* (see Section 6.3). 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[19] OpenDocument Spreadsheet (ODS) file [114]. 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.

Deviations

The experiment execution and the data collection were performed as described in Section 6.4 and Section 6.4. We did not observe any unforeseen difficulties and did not deviate from the experiment plan.

6.5 Analysis

All statistical analysis was performed with the software tool R[20]. The analysis processes[21] contain the following steps: (1) load the prepared data-set from Section 6.5; (2) calculate the descriptive statistics for the dependent variables which are explained in detail in Section 6.5; (3) perform a group-by-group comparison with appropriate statistical hypotheses tests which are explained in detail in Section 6.5; (4) generate table/plot information in order to include this information in this chapter. In order

[19] See `https://www.libreoffice.org` for version `6.1.4.2`.
[20] See `https://www.r-project.org` for version `3.5.2`.
[21] See `analyze.r` at [120].

to reproduce the analysis results, some R library package dependencies have to be installed[22].

Data-Set Preparation

The raw data[23] collected during the experiment execution phase (see Section 6.4) was prepared[24] in the following manner: (1) the obtained LibreOffice ODS file [114] was exported to a Comma-Seperated Values (CSV) file [138]; (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 6.3[25]; (5) and stored as an R Data-Set (RDS) file [127] for further processing and analysis.

Descriptive Statistics

The participants' experience and characteristics (background information) are captured in the experiment by: age (see Figure 6.2a), gender, course, and level of education, programming experience (see Figure 6.2b), modeling experience, software (SW) and hardware (HW) industry experience (see Figure 6.2c), and programming and specification languages used[26]. Overall, the random distribution of the participants to the experiment groups is almost balanced.

The participants' programming experience (see Figure 6.2b) 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 6.1 contains the number of observations, central tendency measures, and dispersion measures per language construct for the dependent variable **Correctness**[27] and this acquired data is visualized as a kernel density plot in Figure 6.3a and a box plot in Figure 6.3b. 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

[22] See `install.r` at [120].

[23] The data-set is published in the long term open data archive Zenodo [120] together with all documents and R scripts.

[24] See `prepare.r` at [120].

[25] For detailed formula, see `prepare.r` Line 235-340 at [120].

[26] See `appendix.pdf` at [120] for supplementary background information.

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

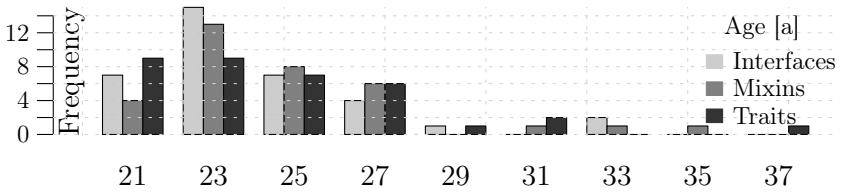
peak at 0.55 and *Mixins* has its peak at 0.45. In contrast to the two other groups, the *Traits* groups has two peaks, one at about 0.215 and the other one at about 0.525.

Table 6.2 contains the number of observations, central tendency measures, and dispersion measures per language construct for the dependent variable **Duration**[28] and this acquired data is visualized as a kernel density plot in Figure 6.4a and a box plot in Figure 6.4b. 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.

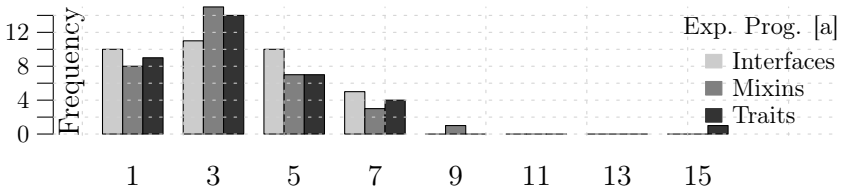
Hypothesis Testing

Due to the presence of three experiment groups and two dependent variables, the Multivariate Analysis of Variance (MANOVA) [21] would be a suitable statistical procedure, but necessary assumptions must be met to apply this method. The investigation of the kernel density plots – Figure 6.3a for **Correctness** and Figure 6.4a for **Duration** – indicates that not all distributions of the experiment groups are

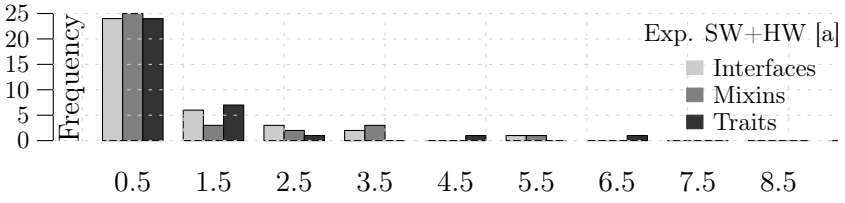
[28]Unit is duration in seconds (denoted [s]).



(a) Participants' Age



(b) Participants' Programming Experience



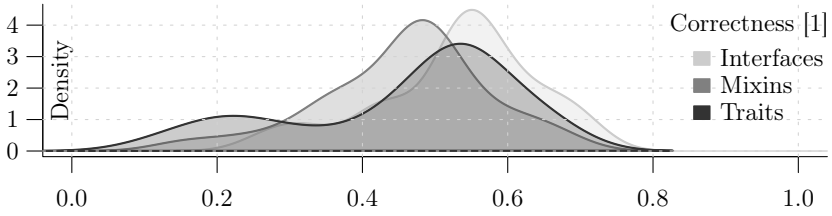
(c) Participants' SW/HW Industry Experience

Figure 6.2: Histograms per Group of Participants' Background Information

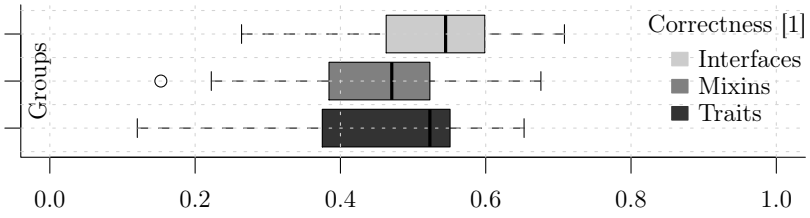
Table 6.1: Descriptive Statistics per Group of **Correctness**

	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

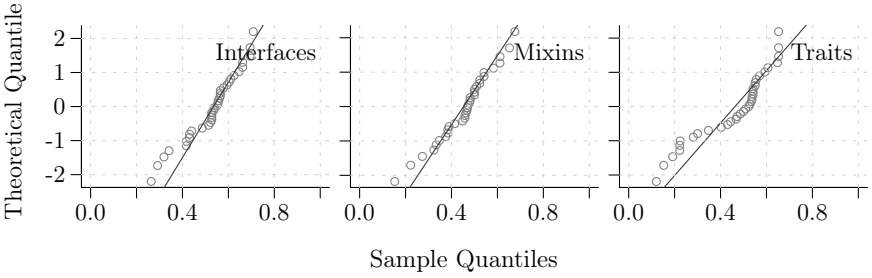
normally distributed, which the MANOVA would need in order to be applied. We applied the Shapiro-Wilk normality test [139] (last row in Table 6.1 and Table 6.2) and only the *Traits* group for the dependent variable **Correctness** shows a significant ($p \leq 0.05$) difference to the normal distribution, 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 normal Q-Q plots for both dependent variables, which are depicted in Figure 6.3c for **Correctness** and Figure 6.4c for **Duration**. All distribution plots indicate that the linearity assumption is not met



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



(b) Box Plot of **Correctness**



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

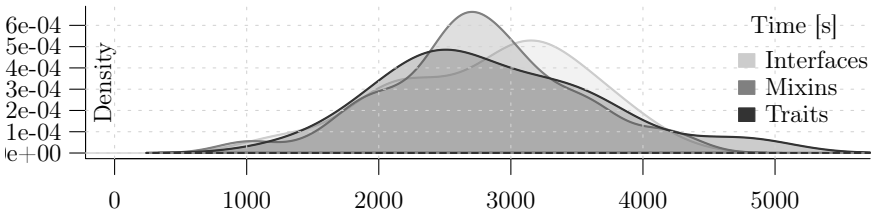
Figure 6.3: Descriptive Plots per Group of the Dependent Variable **Correctness**

Table 6.2: Descriptive Statistics per Group of **Duration**

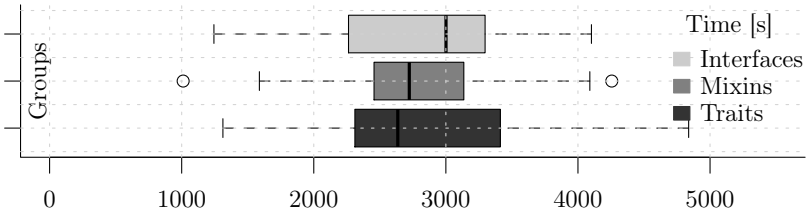
	Interfaces	Mixins	Traits
Observations [1]	35	33	34
Mean [s]	2833.00	2753.33	2856.97
Standard deviation [s]	718.33	702.84	815.27
Median [s]	3001.00	2723.00	2636.00
Median abs. deviation [s]	762.06	612.31	728.70
Minimum [s]	1244.00	1011.00	1312.00
Maximum [s]	4102.00	4256.00	4838.00
Skew [1]	-0.3657	-0.0757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

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.

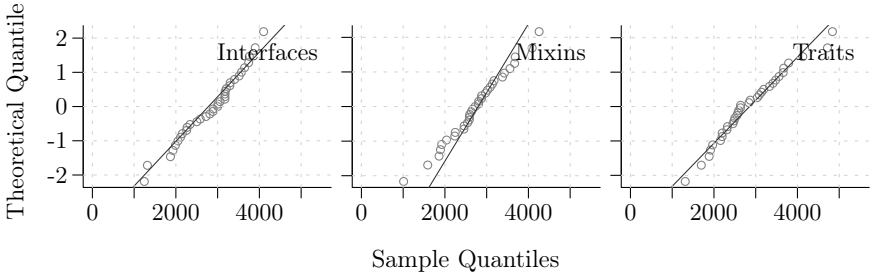
When we considered our acquired data, according to Kitchenham et al. [81], we cannot use the Kruskal-Wallis test [85] because it is strongly affected by unequal variances. Therefore, we select a robust non-parametric test called Cliff's δ [34]. This testing method is unaffected by non-normal data, change in distribution, and (possible) unstable variance.



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



(b) Box Plot of **Duration**



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

Figure 6.4: Descriptive Plots per Group of the Dependent Variable **Duration**

Table 6.3: Hypothesis Tests per Group Combination of **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

The results of the Cliff's δ test is shown in Table 6.3 for the dependent variable **Correctness** and in Table 6.4 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 [46], 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 [14].

According to the FDR adjusted p-values (p_{FDR}) in Table 6.3 and Table 6.4, there is evidence not to reject some null hypotheses of this study (see Section 6.3). Since Cliff's δ test is closely related to the Wilcoxon rank sum test [157] (also know as Mann-Whitney test [98]), 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 Table 6.3 and Table 6.4 along with the appropriate FDR adjusted p-value $p_{W_{FDR}}$.

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*

Table 6.4: Hypothesis Tests per Group Combination of **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

Table 6.5: Correlation per Group of **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

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.

6.6 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

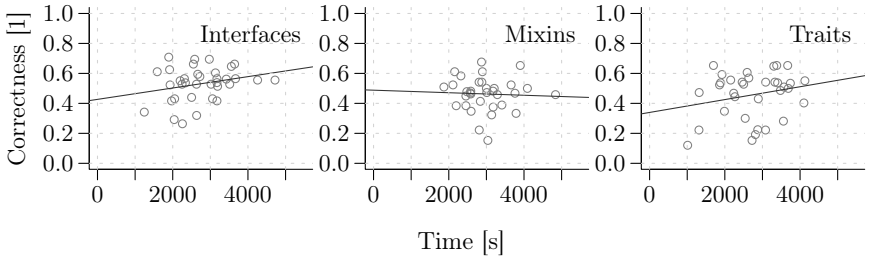


Figure 6.5: Scatter Plot per Group of Variables **Correctness** to **Duration**

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* 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 6.5) and correlation (Table 6.5) 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 Figure 6.6. 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.

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

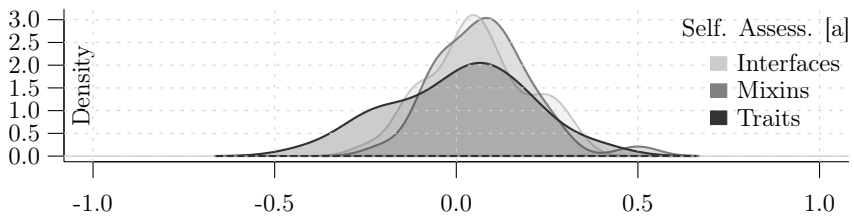


Figure 6.6: Kernel Density Plot per Group of Participants' *Self Assessment*

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 6.3).

Threats to External Validity

A possible threat to external validity is that we carried out the experiment with students as participants because this limits the ability to make generalizations. 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.

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. [71], Czepa and Zdun [40]), but it cannot be ruled out that other constructs would be a better measure for understandability.

Berger et al. [15] 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 [121] 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.

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.

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.

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.

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* [83] 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.

6.7 Conclusion

This chapter 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.

"If it's your decision, it's design; if not, it's a requirement." – Alistair Cockburn

CHAPTER 7

Usability Study

Modern object-oriented languages offer a variety of language constructs to provide easy-to-comprehend abstractions to express structural and behavioral aspects of specifications. Most of them either offer interfaces or traits in addition to classes and inheritance. In this chapter[1], we describe a follow-up study of Chapter 6 about the investigation of object-oriented abstractions such as interfaces and traits for ASM-based specification languages. We report on a controlled experiment with 98 participants to study the specification efficiency and effectiveness in which participants needed to comprehend an informal specification as problem (stimulus) in form of a textual description and express a corresponding solution in form of a textual ASM specification using either interface or trait syntax extensions. The study was carried out with a completely randomized design and one alternative (interface or trait) per experimental group. The results indicate that specification effectiveness of the traits experiment group shows a better performance compared to the interfaces experiment group, but specification efficiency shows no statistically significant differences. To the best of our knowledge, this is the first empirical study studying the specification effectiveness and efficiency of object-oriented abstractions in the context of formal methods.

7.1 Introduction

In 1993, Gurevich [63] described the ASM theory, which is a well-known state-based formal method consisting of transition rules and algebraic functions. It has been used extensively by scientists for a broad research field ranging from software, hardware and system engineering perspectives to specify, analyze, verify, validate, and construct systems in a formal way [129]. ASMs are used to formally describe the evolution of function states in a step-by-step manner[2] and are used to specify sequential, parallel, concurrent, reflective, and even quantum algorithms. Based on the ASM theory by

[1]The content of this chapter is a revised version of the TOSEM'21 article [121].
[2]The ASM theory was formerly called *Evolving Algebra*.

Gurevich [63], several theory improvements and ASM-based language implementations were developed, which were summarized by Börger and Stärk [26] and Börger and Raschke [25]. The diversity of ASM-based applications ranges from formal specification of semantics of programming languages, such as those for Java by Stärk et al. [146] or VHDL by Sasaki [133], compiler back-end verification by Lezuo [90], software run-time verification by Barnett and Schulte [11], software and hardware architecture modeling e.g. of UPnP by Glässer and Veanes [60], to even RISC designs by Huggins and Campenhout [72].

Nowadays, there are several ASM language syntax definitions and tool implementations available like AsmetaL [58], AsmL [65], CASM [91], and CoreASM [50]. AsmetaL and CoreASM offer a rich tool set to analyze and model ASM specifications and provide a Java-based interpreter to execute and simulate the ASM models. AsmL and CASM are compiler oriented language implementations and offer code generation support of modeled ASM specifications. AsmL is based on the .NET framework whereas CASM provides C/C++ code generation and a high performance interpreter as well. Besides the mentioned ASM languages and tools there exists AsmGofer [136] and XASM [3], but those projects are discontinued.

In addition, many other state-based formal methods besides ASMs exist with their own languages and associated tools e.g. Alloy [74], DEVS [32], EFSM [30], Event-B [2], STATEMATE [67], TLA [86], VDM [19], and Z [125].

Problem Statement

For various ASM languages and tools, as well as in most other state-based formal methods, the proposed modeling languages lack easy-to-comprehend abstractions for describing structural and behavioral aspects of specifications in a reusable and maintainable manner. Most of today's specification languages have implemented basic object-oriented abstractions such as classes and inheritance. As there are known problems in such abstractions, leading to complexity, ambiguity, and low comprehensibility, such as the *diamond inheritance problem of multiple inheritance* [99], it would make sense to study more advanced abstractions as well. Today, many modern language implementations restrict class-based language constructs to allow only single inheritance models and add additional abstractions such as interfaces [28] or traits [134] to the language. A prominent example for ASMs is the modeling language AsmL [65] which uses the class abstraction along with a single inheritance model to encapsulate the state and behavior. A similar approach can be observed in the state-based formal methods community. Object-Z [143] or Z++ [88] provide class-based language constructs with inheritance and polymorphism concepts.

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

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

There is a debate in the object-oriented community[3], which of the abstractions, interfaces or traits, is best suited to express behavioral aspects, and many implementations combine different language constructs. A notable example would be the programming language Scala [111], which offers a trait syntax that is similar to the Java [126] interface syntax and offers a class-based implementation and extension syntax. Another example of mixed language constructs, namely interfaces and traits, can be found in the programming language Rust [101], where the language user has to express interface definitions through traits. Empirical research on language constructs in ASM languages and similar state-based formal methods can provide some decision guidance to language designers and compiler engineers on choosing language constructs in specification language designs and implementations. So far such empirical research is rare. Höfer and Tichy [70] analyzed 133 reviewed articles of the Journal of Empirical Software Engineering in the timescale from 1996 to 2006. They have discovered that controlled experiments about formal methods in general are underrepresented and that *"studies about programming languages and programming paradigms are conspicuously absent"*. They further concluded more experiments in this direction would encourage more discussions on the comprehensability of programming languages and formal methods, and eventually improve the language engineering process.

Due to the fact that so far studies about state-based formal methods and the comprehensibility of object-oriented abstractions and language constructs in their context are missing (see Section 7.2), our study also aims to make a contribution to improve the state of empirical knowledge about formal specification languages. Prior to this work, we already have conducted another study [119] and investigated the effects on how language users (experiment participants) understand structural and behavioral aspects of a state-based formal method language (ASM) by reading a given ASM specification as stimuli and answering questions about the properties of given specifications. The provided ASM specifications were represented in three different language constructs – interfaces, mixins, and traits.

Research Objectives, Hypotheses, and Results

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

State-based formal methods and their modeling languages are usually based on core concepts that are significantly different from classes and objects. Reusable and

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

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 [91] [94] [123] [117] as a representative for ASM-based languages and tools. As our study is focused on the general notion of adding object-oriented language constructs to these languages and tools, we believe most of our results can have an impact on other ASM languages. In this study the term **specification effectiveness** corresponds to how well (reading, understanding, and writing) and the term **specification efficiency** corresponds to how fast (duration time of processing) a participant comprehends a given stimuli and specifies an example ASM specification using one of the two object-oriented abstractions. We define the experiment goal using the GQM template [152] as follows: **Analyze** the *Interfaces* and *Traits* object-oriented abstractions (language constructs) **for the purpose of** their evaluation **with respect to** their *specification effectiveness* and *efficiency* **from the viewpoint of** the novice software developer or designer **in the context (environment) of** a moderately advanced university software engineering course. Our hypotheses are influenced by the debate in the object-oriented communities which seems to favor traits over interfaces. We hypothesized that specification effectiveness measured by the dependent variable correctness shows a significantly better performance for traits compared to interfaces as well as that specification efficiency measured by the dependent variable duration shows a significantly better performance for traits compared to interfaces. This hypothesis was influenced by the debate in the object-oriented community, which often discusses traits more favorably than interfaces[4] or points out that "Traits are Interfaces"[5] with code-level reuse functionality. However, it is not obvious whether or not such opinions yield a statistically significant difference, and whether or not they can be mapped to the domain of state-based formal languages. In addition, interfaces are probably the best known abstraction to developers today, and like most ordinary developers our participants are trained in programming languages offering the language construct interfaces in Java or how to model interfaces through a C++ abstract class.

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

Structure of this Chapter

In Section 7.2, we describe object-oriented abstractions, ASMs, the used ASM-based language representations used in this study, and present related studies. Section 7.3 elaborates the planning of this study. In Section 7.4, we describe the execution of the

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

experiment, while the results are presented in Section 7.5 and discussed in Section 7.6. We conclude the chapter in Section 7.7.

7.2 Background

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

Object-Oriented Abstractions

Interfaces define a *protocol* of (typed) operations (signatures) to which an *implementer* of a certain interface (type) must conform [28]. An interface defines a type signature. No behavioral or state information can be defined through interfaces. Each implementer of the interface has to provide an implementation of the complete interface. *Traits* are similar to interfaces with the difference that they can define *stateless behavior* which depends only on the trait itself [134]. Therefore, each implementer can reuse and rely on existing behavioral implementations which is not possible through *Interfaces*. Figure 7.1 depicts both object-oriented abstractions and exemplifies the language construct properties. On the left side, an *Interface* example with two interfaces is shown. $Interface_1$ gets implemented by $Implementer_1$ and $Implementer_2$, whereas $Interface_2$ is only implemented by $Implementer_2$. The same scenario is expressed through the object-oriented abstraction *Traits* on the right side of the figure. As traits can define not only a protocol, the $Trait_1$ directly defines $Behavior_1$ in the trait itself. Thus $Behavior_1$ can be reused by both implementers.

Abstract State Machines

ASMs are used to express calculations in an abstract manner for many different application fields. According to Gurevich and Tillmann [66], the ASM thesis states

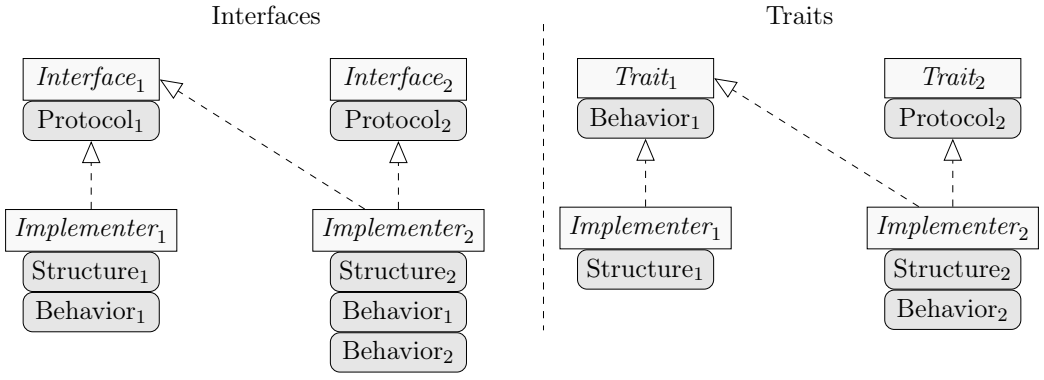


Figure 7.1: Overview of Language Construct Properties

that if there is a computer system A, it can be simulated in a step-by-step manner by a behaviorally equivalent ASM B. The resulting ASM theory and formal method consist of three core concepts: (1) an *ASM specification* language, which looks similar to pseudo code to express rule-based computations over algebraic functions with arbitrary data structures and type domains; (2) a *ground model* serving as a rigorous form of blueprint and reference model; and (3) incremental *refinement* of the reference model by instantiating more and more concrete models which uphold the properties of the reference model [26].

ASMs has two fields of works – *modeling* and *refinement*. In order to model an application or system through an ASM specification, an ASM *language user* has to understand the three most important modeling concepts [25] of ASMs:

States are the notion in ASMs to define the objects and attributes of an application or system through relations and function types. Therefore, all state information in an ASM specification is expressed through a *function* definition (see Section 7.2).

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*. Since ASM rules allow interleaved parallel and sequential execution semantics [64], a *correct* ASM specification does not allow the *update* (insertion to the *update-set*) of the same *function location* twice or more with a different value, which is referred in the literature as an *inconsistent update* [25]. A *language user* can model transactions though *named rule* definitions (see Section 7.2).

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

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

A more detailed description and elaboration of the ASM modeling and refinement concepts is given by Börger and Raschke [25].

ASM Language Representation

In this study, we use the basic syntax elements from the CASM language[6] [117]. The CASM language elements used can be found in a similar fashion in other ASM languages; hence, we believe it is likely that our results can be applied to other ASM languages. CASM is a statically typed ASM-based specification language. Every specification is composed of definition elements. Relevant to this study are the following three definitions – *Function*, *Derived*, and *Rule* definitions.

Function Definition A `function` definition specifies an n-dimensional state (argument types) which maps to a certain function type (return type). E.g. variables in a programming language are modeled as nullary *functions* in ASMs, or hash-maps can be expressed as unary *functions* in ASMs. Listing 7.1 illustrates the concrete syntax and some examples.

Derived Definition A `derived` definition specifies functions which state values can only be derived from other *functions* or *deriveds* without modifying the ASM *state*. Therefore, *derived* functions are side-effect free. Listing 7.2 illustrates the concrete syntax and some examples which use state information from Listing 7.1.

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 expressed through basic ASM *rules* namely: (1) *update* rule to produce

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

```
1 function counter : -> Integer // variable
2
3 function personsAge : String -> Integer // hash-map
```

Listing 7.1: *Function* Definition Example

```
1 derived nextCounter -> Integer = counter + 1
2
3 derived isFullAged( name : String ) -> Boolean =
4   ( personsAge( name ) >= 18 )
```

Listing 7.2: *Derived* Definition Example

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

Listing 7.3: Named *Rule* Definition Example

a new value for a given state function (*location*); (2) *block* rule to express bounded parallelism of multiple *rules*; (3) *sequential* rule to express sequential execution semantics of multiple *rules*; (4) *conditional* rule to specify branching (`if-then-else`); (5) *forall* rule to express parallel computations; (6) *choose* rule to specify nondeterministic choice; (7) *iterate* rule to express iterations; and (8) *call* rule to invoke *named rules* (sub-rule call).

A more detailed explanation of all ASM *rules* is given by Börger and Raschke [25]. Listing 7.3 illustrates the concrete syntax and an example which depends on some definitions from Listing 7.1 and Listing 7.2.

Experiment Language Construct Representations

Besides a class concept used in AsmL [65], no other object-oriented language construct has been introduced in the ASM language and tool landscape. To enable moving the state-of-the-art in advanced object-oriented abstractions for such formal languages forward, this study tests two language construct representations, namely interfaces and traits, to search for a suitable object-oriented abstraction to structure state and behavioral aspects for such languages in general and specifically for CASM. In order to do so, we introduced three new definitions for this study into the existing CASM syntax – *Feature*, *Structure*, and *Implement* definitions.

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

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

Implement Definition An `implement` definition specifies which *feature* gets implemented and used by which *structure*.

This definition element binds default or extended functionalities (behaviors) to a certain type (structure).

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

Language Construct Interfaces (Experiment Group A) The `feature` syntax in the language construct *Interfaces* only describes the *protocol* consisting of the set of operations [97] [28] 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 7.4 depicts an example specification with the *Interface* language construct[7]. This syntax is primarily influenced by the Java programming language [126] interface syntax.

Language Construct Traits (Experiment Group B) The `feature` syntax in the language construct *Traits* is equal to *Interfaces* except that it supports defi-

[7] See `form_ifaces.pdf` at [122].

```
1  feature Formatting = {
2    derived toString : -> String
3  }
4
5  structure Person implement Formatting = {
6    function name : -> String
7    function age : -> Integer
8
9
10
11   derived getName -> String = this.name
12   derived getAge -> Integer = this.age
13
14   rule setName( name : String  ) = this.name := name
15   rule setAge( age : Integer ) = this.age := age
16
17
18
19   // encapusalted feature implementation
20   derived toString -> String =
21     this.getName() + ( this.getAge() as String )
22 }
```

Listing 7.4: *Interfaces*-Based Example Specification

```
1  feature Formatting = {
2    derived toString -> String
3  }
4
5  structure Person = {
6    function name : -> String
7    function age  : -> Integer
8  }
9
10 implement Person = {
11   derived getName -> String = this.name
12   derived getAge -> Integer = this.age
13
14   rule setName( name : String  ) = this.name := name
15   rule setAge( age : Integer ) = this.age := age
16 }
17
18 // decoupled feature implementation
19 implement Formatting for Person = {
20   derived toString -> String =
21     this.getName() + ( this.getAge() as String )
22 }
```

Listing 7.5: *Traits*-Based Example Specification

nition of optional default implementations inside the `feature` definition itself. A `structure` only contains the state information. The behavior in the *Traits* abstraction is implemented through two different kinds of separated `implement` definitions: (1) describes the behavior of the structure; (2) describes the behavior of a certain `feature for` a structure. It is important to note here that a default implementation provided in the `feature` syntax can be overwritten in the `implement` definition. Listing 7.5 depicts an example specification with the *Traits* language construct[8]. This `feature` and `implement` syntax is influenced by the Rust programming language [101] trait syntax[9].

Related Studies

So far, interfaces and traits have mainly been studied in the context of programming languages and mainly by proposing new solutions. A small number of empirical studies exists in this field which are mainly case studies. For instance, Murphy-Hill et al. present a case study on the potential of traits to reduce code duplication [108]. However, so far no study comparing the two language constructs interfaces and traits covered in our study exists and also no controlled experiments.

Interface abstractions have been extensively studied in the context of formal methods [33] [41] [31] and architecture description languages that offer formal representations [112] [59]. Traits in contrast have not yet been studied in the context of formal methods. We are not aware of any formal method that unifies or integrates the two object-oriented language constructs covered in our study.

Overall formal methods have been studied before in only a few empirical studies other than case studies. An example of the few existing studies is the one by Sobel and Clarkson, who study the aiding effect of first-order logic formalisms in software development [145]. Czepa and Zdun [40] and Czepa et al. [39] have studied the understandability of formal methods for temporal property specification using similar research methods as used in this study.

Snook and Harrison [144] performed structured interviews with formal method users asking them about scalability, understandability, and tool support issues. A very interesting aspect of this study is that the participants report that *"the precise and accurate nature of the specification makes the coding task straightforward and the coder is less likely to build in redundant code."* [144]. Another interesting finding in this study is that the *"interviewees thought that the difficulties with using formal specifications were in finding the useful abstractions from which to create models."* [144]. Snook and Harrison [144] argue that the problem behind the interviewees statement is that programming languages mainly focus on structural aspects first whereas formal methods focus on behavioral aspects.

[8] See `form_traits.pdf` at [122].

[9] See `https://doc.rust-lang.org/rust-by-example/trait.html` for Rust's trait syntax.

We are not aware of any empirical study systematically investigating object-oriented language constructs in the context of state-based formal methods. Only, in our own prior work we conducted a study [119] with 105 participants where we analyzed how well experiment participants understand given ASM specifications which are represented in three different language constructs – interfaces, mixins, and traits. The results of this experiment showed that the object-oriented abstractions interfaces and traits are better understandable than mixins.

7.3 Experiment Planning

This study is structured following the guidelines by Jedlitschka et al. [77] on how empirical research shall be conducted and reported in software engineering. Moreover, the guidelines by Kitchenham et al. [82], Wohlin et al. [158], and Juristo and Moreno [80] 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. [81].

Goals

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

Context and Design

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

Participants

All 98 participants of the experiment are BSc students of the Faculty of Computer Science at the University of Vienna, Austria enrolled in the course Software Engineering

2 (SE2)[10] in the winter term 2018/19. The BSc students enrolled in the SE2 course are used as proxies for novice to moderately advanced software architects, designers, or developers. This course, which is a mandatory part of the BSc curricula at the University of Vienna, is intended for students in the fourth semester of the BSc curricula. The content of this course is about teaching principles of the construction and design of software systems, investigating different methods and tools, design patterns, programming styles, and how to tackle non-functional requirements. The participants (students) received training in programming, software engineering, (data) modeling, basic formal methods, algorithms, and mathematics in previous courses.

At the beginning of the SE2 course, the students were informed that during the semester there will be an opportunity to participate in an experiment. The attendance of the experiment was optional, and the submitted solutions (filled out survey forms) were rewarded with up to 6 bonus points. There was the option to receive the 6 bonus points by performing the tasks, but not participate in the experiment (opt out option). How well (correctness, see Section 7.5) a participant answered the survey determined the bonus points. In total, there were 98 participants, which were randomly allocated to the treatments (using one of the two language construct representations in an ASM specification language, see Section 7.2). Due to random assignment of the participants to groups – *Interfaces* (Group A) and *Traits* (Group B) – the final distribution resulted in 49 : 49. Some may argue that students as experiment participants are not good proxies for novice software engineers. The experiment participants are students of an advanced course (SE2) at the University of Vienna, which trained the students in abstractions needed for the experiment task domain, and were trained in basic formal methods in prior courses. Easy to understand formalisms are key to correct specifications in practice. We expect advanced students to be good proxies for inexperienced developers and architects.

In this study, we do not focus on well trained experts as they are usually also much better trained in formalisms, because the goal of the study is not to focus on techniques that can only be applied by a few very well trained experts. Furthermore, according to Kitchenham et al. [82] using students *"is not a major issue as long as you are interested in evaluating the use of a technique by novice or nonexpert software engineers. Students are the next generation of software professionals and, so, are relatively close to the population of interest"*. This is directly reflected in this study because some of the students who participated in the experiment show several years of programming experience as well as several years of work experience in the software and/or hardware industry (see Figure 7.2d). Other studies by Svahnberg et al. [151] or Salman et al. [132] would argue even further and state that under certain circumstances, students are valid representatives for professionals in empirical software engineering experiments.

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

Material and Tasks

The experiment is based on a selection of basic software system applications. The selection includes a *Calculator System*, an *Event Scheduling/Pooling System*, and a *Traffic Control System* as example applications inspired by some examples provided by Börger and Raschke [25].

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

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

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

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

(1) Information Sheet An experiment information document[11] explaining the ASM language syntax and semantics **without the experiments' language construct** syntax and semantics extensions.

(2) Survey Form Two experiment survey forms[12] per experimental group and language construct containing the actual survey along **with the explicit experiments' language construct** syntax and semantics extension and description **per experimental group**.

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

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

[11] See `info.pdf` at [122].
[12] See `form_ifaces.pdf` and `form_traits.pdf` at [122].

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

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

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

Variables

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

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

The precise description on how the correctness is computed is given in Section 7.5.

(2) Duration The specification efficiency (duration) is the time it took the participants to comprehend the informal specification (stimuli) and model a corresponding ASM specification by using one of the two object-oriented abstractions. Important to note here is that the measurement of the duration variable only includes the processing time (reading, comprehending, and writing) and excludes breaks (see Section 7.3).

Hypotheses

We hypothesized that *Traits* are easier to comprehend than *Interfaces* due to the fact that *Traits* have the ability to avoid code duplication and clearer separation of state and behavioral aspects by having almost equal API declaration styles as *Interfaces*. Consequently, as suggested by Wohlin et al. [158] we formulate the following null hypotheses, where specification effectiveness is measured by the correctness variable and specification efficiency is measured by the duration variable:

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

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

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

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

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

7.4 Experiment Execution

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

Preparation

Two weeks before the experiment we handed out the preparation material (the experiment *information sheet*, see Section 7.3) through an e-learning platform[13]. This document provided general information of the upcoming experiment and an introduction to the ASM language syntax and semantics used without explaining one of the two language constructs. All ASM language concepts used are depicted with short example ASM specification snippets. The participants were allowed to use this document during the experiment in printed form. The main reason why we provided the experiment information document is that all participants needed to be educated to the same level of detail with regard to a state-based formal method and specifically to a concrete ASM language representation (see Section 7.2).

Procedure

The experiment was carried out using paper and pencil, as if it were an (closed book) exam. Participants were allowed to bring only one aid – the *information sheet* – to process the experiment survey form as described in the previous Section 7.4. At the beginning of the experiment, every participant received a random experiment *survey form* (see Section 7.3). They were instructed to fill out and process the survey from the first page to the last page in this particular order. Furthermore, a clock with seconds granularity was projected onto a wall to provide timestamp information to the participants. They were asked to track start and stop timestamps during the processing of the experiment tasks. After the experiment every participants' modeled ASM specification was examined through a list of evaluation criteria (see Section 7.5) and the results of the examination was recorded in a spreadsheet. The participants' task start and stop timestamps were converted to a duration in seconds and summed

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

up to a total duration for all tasks. We used the four-eyes principle during every manual work step (answer obtaining and timestamp conversion) in the data collection. The experiment execution and data collection were performed as described in this section and we have not observed any form of deviations or unforeseen difficulties.

7.5 Analysis

All statistical analysis was performed with the software tool R[14]. The analysis processes[15] contain the following steps: (1) load the prepared data-set from Section 7.5; (2) calculate the descriptive statistics for the dependent variables which are explained in detail in Section 7.5; (3) perform a group-by-group comparison with appropriate statistical hypotheses tests which are explained in detail in Section 7.5; (4) generate table/plot information in order to include this information in this chapter. In order to reproduce the analysis results, some R library package dependencies have to be installed[16].

Data-Set Preparation

The raw data[17] collected during the experiment execution phase (see Section 7.4) was prepared[18] in the following manner: (1) the obtained LibreOffice ODS file [114] was exported to a CSV file [138]; (2) the CSV file was imported for further processing; (3) type castings of several data rows were performed; (4) the calculation of task-based and overall **Duration** times; (5) the calculation of task-based and overall **Correctness** values; and (6) stored as an RDS file [127] for further processing and analysis.

The calculation of the **Correctness** value is composed out of a check list of yes-and-no statements[19] for all the different tasks in the experiment survey forms (see Section 7.3). This list of yes-and-no statements was derived before the experiment execution by specifying ground truth models for both object-oriented language abstractions variants – interfaces and traits – of the informal described experiments' example software application systems. In order to enable a flexible way to compare the participants' solutions from the experiment, the obtained list of yes-and-no statements reflects generic properties the provided and specified models by the participants shall contain. The yes-and-no statements are grouped into five evaluation criteria (categories) – structure, behavior, syntax, reusability, and functionality. The following list depicts for each of the evaluation criteria an example yes-and-no statement:

[14] See `https://www.r-project.org` for version `3.5.2`.
[15] See `analyze.r` at [122].
[16] See `install.r` at [122].
[17] In order to enable reproducability of our results, the data-set (`README.ods`) is made public in the long term open data archive Zenodo [122] together with all documents and R scripts.
[18] See `prepare.r` at [122].
[19] See `README.ods` for the complete list of the yes-and-no statements along with the collected data for all participants at [122].

Table 7.1: Number of Yes-and-No Statements per Evaluation Criteria and Tasks

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

(1) Structure Did the participant specify certain structural elements? An example structural evaluation criteria statement for Task 1[20] is defined as follows: *"Proxy structure defined"*?

(2) Behavior Did the participant specify certain behavioral elements? An example behavioral evaluation criteria statement for Task 1[20] is defined as follows: *"Client implemented default behavior"*?

(3) Syntax Did the participant use the correct language construct syntax for the assigned treatment? An example syntactical evaluation criteria statement for Task 1[20] is defined as follows: *"Server valid abstraction syntax"*?

(4) Reusability Did the participant recognized reusable elements and did (s)he specify it through the correct language construct syntax for the assigned treatment? An example reusable evaluation criteria statement for Task 1[20] is defined as follows: *"Operations implemented for Proxy"*?

(5) Functionality Did the participant specify certain functionalities? An example functional evaluation criteria statement for Task 1[20] is defined as follows: *"Message provides unique identification"*?

In total there exist 65 yes-and-no statements per experiment participant. By accumulating the percentage value of all yes-and-no statements a total of 100% **correctness**[21] can be achieved. Table 7.1 depicts the number of yes-and-no statements in total and the dissection per evaluation criteria and tasks.

Descriptive Statistics

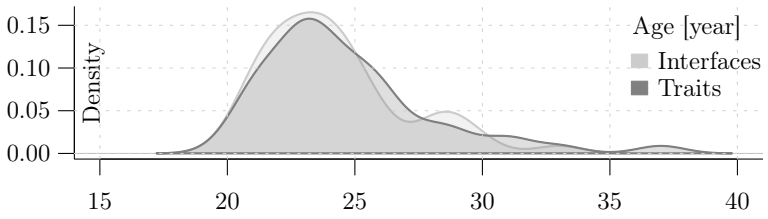
Background Information: The participants' experience and characteristics are captured in the experiment through eight parameters[22] and the results indicate that overall, the random distribution of the participants to the experiment groups is almost balanced. The participants' age (see Figure 7.2a) shows a similar distribution for both groups with a peak around 23 years. The programming experience of the participants measured in years (see Figure 7.2b) indicate that the interfaces group has a more than

[20]See `form_ifaces.pdf` or `form_traits.pdf` for description of Task 1 at [122].
[21]See `prepare.r` Line 97-250 at [122] for a detailed formula.
[22]See `appendix.pdf` at [122] for more detailed supplementary background information.

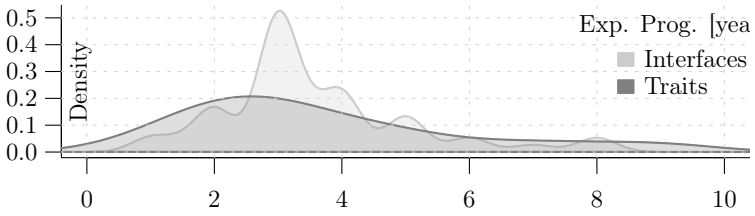
twice higher density around 3 years of experience in programming compared to the traits group which has its peak around 2.5. This is the only background information parameter showing a slightly unbalanced distribution and indicates that the general programming experience level is higher in the interfaces experiment group. This discrepancy is attributed to the randomized distribution of the experiment survey to the participants.

In contrast to the programming experience, the distribution of the participants' specification (modeling) experience measured by years (see Figure 7.2c) is quite similar for both groups with a peak at 2 years. Since our participants are students, the peak of the software (SW) and hardware (HW) industry experience measured in years (see Figure 7.2d) is at zero years, but a number of students show a similar level of industry experience between 1 to 3 years.

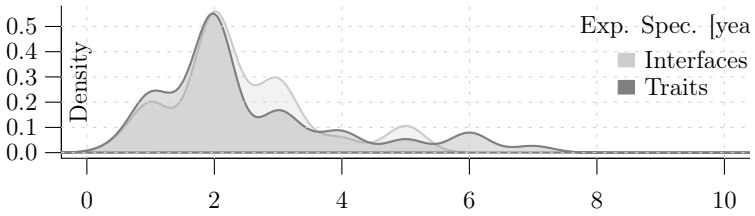
The experiment total ratio between female and male participants is 37 (37.76%)



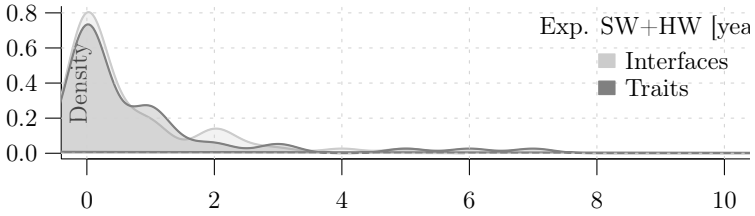
(a) Kernel Density Plot of Participants' Age



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



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



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

Figure 7.2: Descriptive Plots per Group of Participants' Background Information

Table 7.2: Participants' Gender

Gender	Interfaces	Traits
Female	20	17
Male	29	32

Table 7.3: Participants' Level of Education

Education	Interfaces	Traits
None	42	45
BSc	7	4

Table 7.4: Participants' Programming Language Knowledge

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

Table 7.5: Participants' Prior Knowledge of Formal Methods

Interfaces	Traits
5	4

: 61 (62.24%). The interfaces group has 20 (40.82%) female and 29 (59.18%) male participants and the traits groups has 17 (34.69%) female and 32 (65.31%) male participants.

From the perspective of prior computer science education (see Table 7.3) only 11 (11.22%) students have a previous BSc degree and the other 87 (88.78%) participants are undergraduates. The numbers are quite comparable in the two experiment groups. All participants (100%) are familiar with Java and 94 (95.92%) participants – 46 (93.88%) interfaces group and 48 (97.96%) traits group – are familiar with C++. That means the interface abstraction should be more than familiar to both experimental groups. We can further observe languages offering traits, besides the programming language PHP (total 80 (81.63%) – interfaces group 41 (83.67%) and traits group 39 (79.59%)), are rather underrepresented in both experimental groups. This is the case for the programming languages Scala (total 27 (27.55%) – interfaces group 11 (22.45%) and traits group 16 (32.65%)), Swift[23] (total 10 (10.20%) – interfaces group 7 (14.29%) and traits group 3 (6.12%)), and Rust where only one of all participants (interfaces group 2.04%) is familiar with the language.

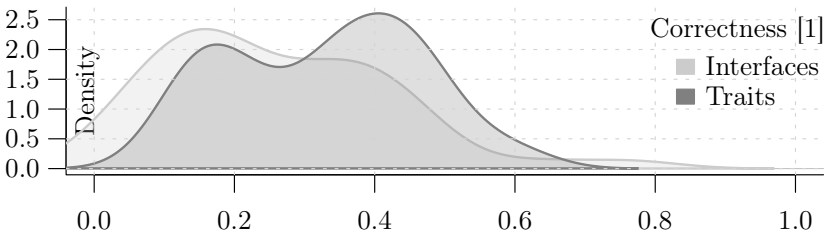
[23]Swift has implemented traits through the *protocol extension* syntax. See, e.g. `https://docs.swift.org/swift-book/LanguageGuide/Extensions.html`.

A very important parameter of the background information is if there are participants which have a prior knowledge of formal methods (see Table 7.5). Accordingly to the obtained results, only 9 participants (9.18%) in total – interfaces group 5 (10.20%) and traits group 4 (8.16%) – have stated that they have prior knowledge in a formal method.

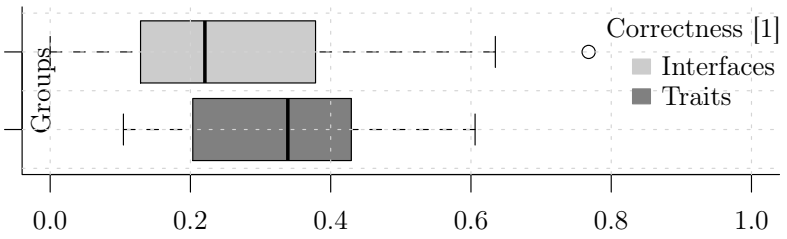
Dependent Variable Correctness: Table 7.6 contains the number of observations, central tendency measures, and dispersion measures per language construct for the dependent variable **Correctness**[24] and this acquired data is visualized as a kernel density plot in Figure 7.3a and a box plot in Figure 7.3b. In the box plot we can observe that the median of the *Interfaces* group is almost at the lower quartile value of the *Traits* group. There is one outlier in the *Interfaces* group which performed very well.

The distribution of the *Interfaces* group is left skewed whereas the *Traits* group is right skewed. The *Traits* group has no outlier at all. According to the kernel density

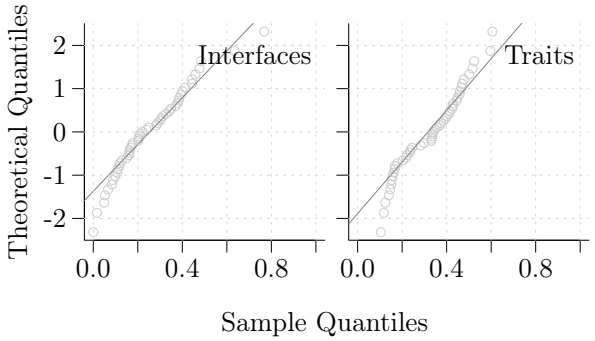
[24]Unit is correctness rate between 0.0 and 1.0 (denoted [1]).



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



(b) Box Plot of **Correctness**



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

Figure 7.3: Descriptive Plots per Group of **Correctness**

Table 7.6: Descriptive Statistics per Group of **Correctness**

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

Table 7.7: Hypothesis Tests per Group Combination of **Correctness**

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

plot, the data does not appear to be normally distributed, and both distributions look different, which implies unequal variances and both distributions have two peaks as well. The *Interfaces* group has one peak at 0.16 and another one at 0.37 whereas the *Traits* group has one peak at 0.17 and another one at 0.41.

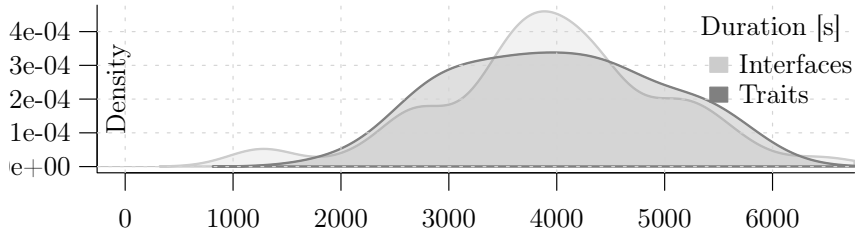
Dependent Variable Duration: Table 7.8 contains the number of observations, central tendency measures, and dispersion measures per language construct for the dependent variable **Duration**[25] and this acquired data is visualized as a kernel density plot in Figure 7.4a and a box plot in Figure 7.4b. In the box plot we can observe that for both groups the median is almost the same (*Interfaces* at 3935 and *Traits* at 3980), but the lower and upper quantiles of the *Traits* group indicate a wider distribution which is reflected in Figure 7.4a. The latter shows the data does not appear to be normally distributed for the *Interfaces* group and almost for the *Traits* group, and the two distributions look different, which implies unequal variances. The *Interfaces* group has its peak at 3950 seconds and the *Traits* group has its peak at 4000 seconds. Moreover, the box plot shows three outliers for the *Interfaces* group – two participants which processed the experiment (*survey form*) really fast and one participant who

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

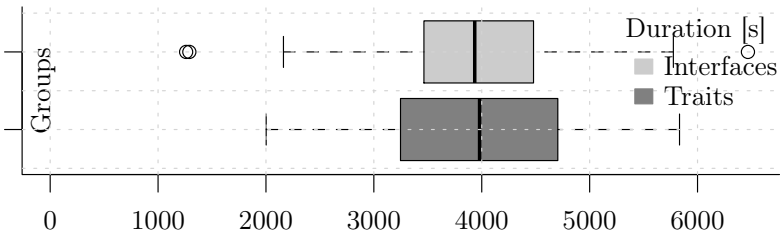
processed it really slow.

Hypothesis Testing

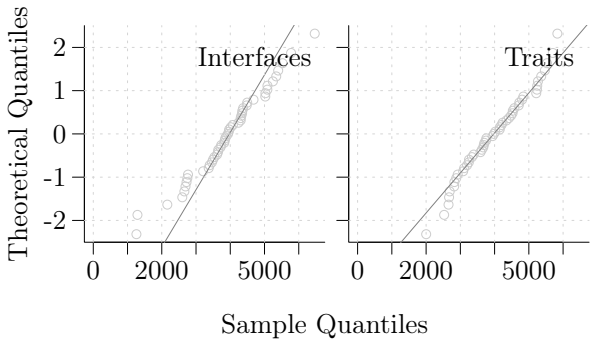
Due to the presence of two experiment groups and two dependent variables, the MANOVA [21] would be a suitable statistical procedure, but necessary assumptions must be met to apply this method. The investigation of the kernel density plots – Figure 7.3a for **Correctness** and Figure 7.4a for **Duration** – 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 [139] (last row in Table 7.6 and Table 7.8) and for both groups (*Interfaces* and *Traits*) for the dependent variable **Correctness** shows a significant ($p \leq 0.05$) difference to the normal distribution, which would make MANOVA not suitable for **Correctness** but suitable for **Duration**. To finally conclude that the MANOVA method cannot be applied, we visually inspected the normal Q-Q plots for both dependent variables, which are depicted in Figure 7.3c for **Correctness** and Figure 7.4c for **Duration**.



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



(b) Box Plot of **Duration**



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

Figure 7.4: Descriptive Plots per Group of **Duration**

Table 7.8: Descriptive Statistics per Group of **Duration**

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

Table 7.9: Hypothesis Tests per Group Combination of **Duration**

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

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. [81], we cannot use the Kruskal-Wallis test [85] because it is strongly affected by unequal variances. Therefore, we select a robust non-parametric test called Cliff's δ [34]. This testing method is unaffected by non-normal data, change in distribution, and (possible) unstable variance.

The results of the Cliff's δ test is shown in Table 7.7 for the dependent variable **Correctness** and in Table 7.9 for the dependent variable **Duration**. Due to the fact that we applied this hypothesis test two times, we are required to lower the significance level in order to avoid Type I errors, which is about not detecting an effect that is not present. A suitable approach would be to apply the Bonferroni correction [46], which suggests to lower the current significance level $\alpha = 0.05$ divided by the times a certain test was applied ($n = 2$), which would result into $\alpha' = \frac{\alpha}{n} = \frac{0.05}{2} = 0.025$. Unfortunately, this significance level correction is known to increase Type II errors, which is about not detecting an effect that is present. Therefore, we choose a more

robust correction method which does not increase Type II errors, namely the FDR adjusted p-values [14]. According to the FDR adjusted p-values (p_{FDR}) in Table 7.7 and Table 7.9, there is evidence to reject one of the hypotheses of this study (see Section 7.3). For the dependent variable **Correctness** we found evidence of a better specification effectiveness of expressing structural, behavioral, syntactical, reusable, and functional aspects through ASM specifications from a given informal description of software system applications. The test results on **Correctness** are significant with a small effect size magnitude [81] for the comparison of *Interfaces* and *Traits*, which suggests to reject $\mathbf{H_{0,1}}$ and to accept $\mathbf{H_{A,1}}$. For the dependent variable **Duration** the null hypothesis $\mathbf{H_{0,2}}$ cannot be rejected as the test results are not significant. Therefore, the alternative hypothesis $\mathbf{H_{A,2}}$ cannot be accepted.

7.6 Discussion

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

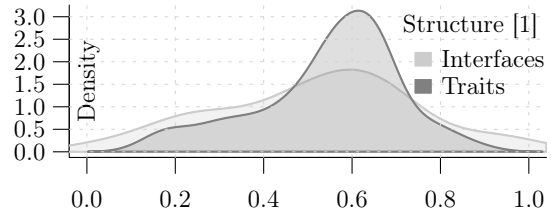
Evaluation of Results and Implications

The descriptive statistics do directly favor one of the language constructs, because by looking at the dependent variable **Correctness**, *Traits* performs better than *Interfaces*. The median of the **Correctness** variable is for language construct *Interfaces* 22.06% and *Traits* 33.89%. Due to the fact that all participants have almost no prior knowledge ($< 10\%$) of ASMs and formal methods in general (checked by an informational question in the survey, see Section 7.5), a median for the specification effectiveness (correctness) between 22% to 34% can be considered a rather good result in this study. For the **Duration** descriptive statistical results, *Interfaces* and *Traits* seem to have a similar distribution. The median of the **Duration** variable is for language construct *Interfaces* $3935s$ ($1h\ 5min\ 35s$) and *Traits* $3980s$ ($1h\ 6min\ 20s$), which are good results in the scope of the processed survey and the achieved **Correctness** results with a limited experiment time of $120min$ ($2h$). Note that the highest participant duration was $6467s$ ($1h\ 47min\ 47s$).

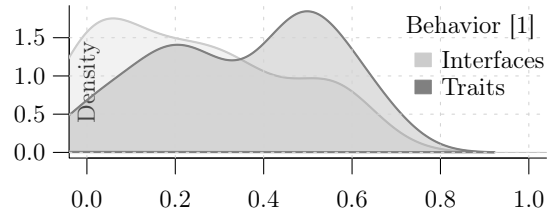
In the inferential statistics *Traits* show a significantly better performance than *Interfaces* in terms of **Correctness** (specification effectiveness). This significance implies that for the ASM language user (novice software developer or designer) it is easier and more effective to express informal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

In order to explain and gain more details about the better **Correctness** results for the *Traits* group compared to the *Interfaces* group, we have dissected the correctness to the five evaluation criteria (see Section 7.5) and analyzed them individually.

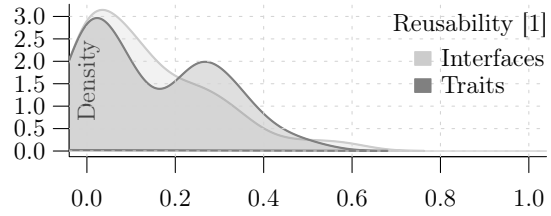
The structural correctness (see Figure 7.5a) value shows a density about twice as high for the *Traits* group with a peak correctness value for both groups around



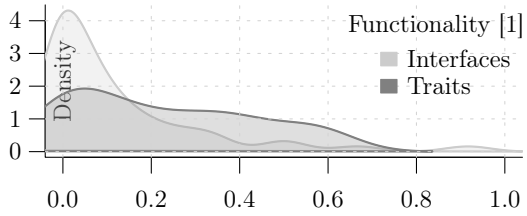
(a) Kernel Density Plot of Structural Correctness



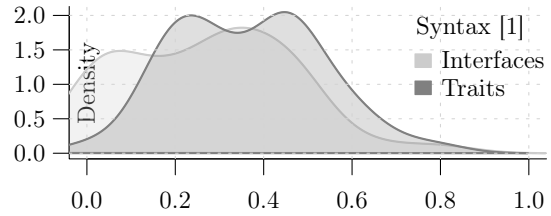
(b) Kernel Density Plot of Behavioral Correctness



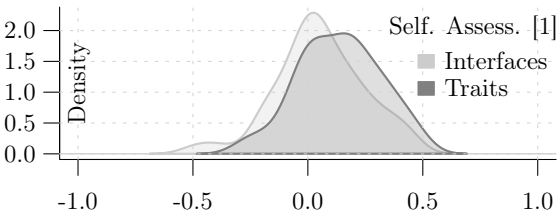
(c) Kernel Density Plot of Reusability Correctness



(d) Kernel Density Plot of Functionality Correctness



(e) Kernel Density Plot of Syntax Correctness



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

Figure 7.5: Descriptive Plots per Group of Correctness and *Self Assessment*

Table 7.10: Correlation per Group of **Correctness** to **Duration**

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

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

The distributions of the functionality correctness (Figure 7.5d) show that a large number of participants of the *Interfaces* group were not able to express functionalities very well. The *Traits* group, in contrast, shows a very stretched distribution from 0% up to 65%. Apparently the participants were able to express (non object-oriented related) functionalities better through the *Traits*-based ASM syntax extension. Figure 7.5e compares syntactical correctness results. We can observe that both groups' distribution have two peaks – 7% and 35% for the *Interfaces* group, and 21% and 45% for the *Traits* group.

The kernel density plot for the participants' *self assessment* is depicted in Figure 7.5f. The self assessment was measured by calculating the difference between the actual **Correctness** value and the participants **Confidence** value that a certain solution to a task they worked on was correct. A self assessment value ≤ 0 means the participant overestimated and ≥ 0 means the participant underestimated the **Correctness** of the given experiment answers. Both experiment groups show almost a similar *self assessment* with its peak in the underestimated section. This implies that both object-oriented abstractions show a similar participants' self assessment regarding their **Confidence** in the **Correctness** of their given solutions.

Studying the scatter plot (Figure 7.6), Spearman's rank correlation, and Pearson

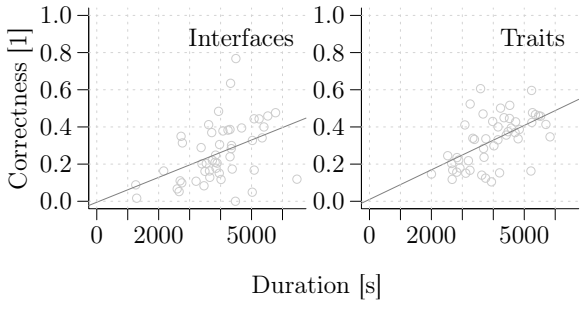


Figure 7.6: Scatter Plot per Group of Variables **Correctness** to **Duration**

product-moment correlation (Table 7.10) of the two dependent variables **Correctness** and **Duration**, we cannot observe a clear (linear nor a non-linear) monotonic trend that the dependent variables are strongly correlated somehow.

As described in Section 7.3 we also asked the participants to fill in a post experiment questionnaire where they could provide us answers using six Likert-scale [95] questions ($\mathbf{Q_n}$) with five possible answers: (1) strongly agree, (2) agree, (3) neutral, (4) disagree, (5) strongly disagree. The questions and their corresponding results are:

$\mathbf{Q_1}$ *"Every given specification was easy to read and understand."* According to the obtained answers (see Table 7.11a), the perceived difficulty was almost equal. This means that most of the participants in both groups agree that the provided informal descriptions of the software system applications were easily understood.

$\mathbf{Q_2}$ *"I had no trouble to specify structural elements of the given informal specifications."* The results in Table 7.11b show that for the *Traits* group 17 (34.69%) participants rank their expressing of structural properties neutral. Among the other participants, one half tends to strongly agree and the other half to strongly disagree. The *Interfaces* group answers of $\mathbf{Q_2}$ are more split with the two biggest groups saying they agree and the other one disagrees.

$\mathbf{Q_3}$ *"I had no trouble to specify behavioral elements of the given informal specifications."* The answers of this question (see Table 7.11c) reflect that in both language construct groups the participants had more or less troubles to express behavioral properties, but the results of the behavioral correctness (see Figure 7.5b) show clearly that the *Traits* group performed way better than the *Interfaces* group.

$\mathbf{Q_4}$ *"I had no trouble to specify functionality extensions for the given informal specifications."* Similar to the answers of $\mathbf{Q_3}$, Table 7.11d shows that the participants of the *Traits* group perceived that they had troubles to express functionality extensions (reusable protocol and behavioral properties) but the results for the correctness values of reusability (see Figure 7.5c) indicate that the *Interfaces* group performed worse than the *Traits* group.

$\mathbf{Q_5}$ *"I am familiar with the language concept called Interfaces."* Accordingly to the participants' background information (see Table 7.4), 100% of them know Java which is more or less reflected in the results to this question (see Table 7.11e), where we asked the participants if they are familiar with the language construct interfaces.

$\mathbf{Q_6}$ *"I am familiar with the language concept called Traits."* In contrast to $\mathbf{Q_5}$, the results of this question (see Table 7.11f) are surprising, because more participants of the *Interfaces* group know the language concept traits compared to the *Traits* experimental group itself. So seemingly the good results for traits have been achieved, even though more knowledge on traits was present in the interfaces group.

Table 7.11: Questionnaire Results $\mathbf{Q_n}$

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

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

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

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

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

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

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

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

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

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

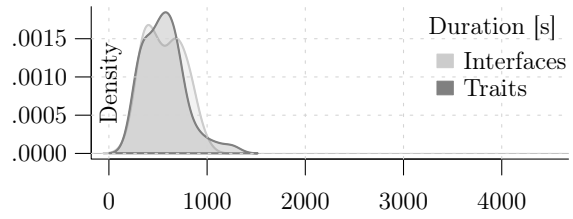
(f) Results of $\mathbf{Q_6}$ (Traits)

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

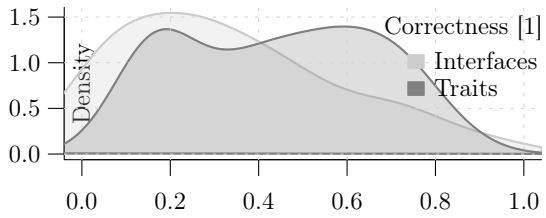
In summary, the post experiment questionnaire shows that the participants believe they understood the constructs to be used reasonably well, and as expected interfaces are better known than traits before the experiment. In this light, our results indicating better results for traits are even more remarkable.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

Exploration of Moderating Variables

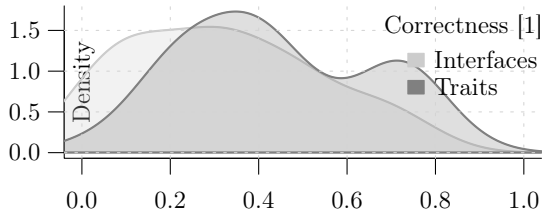
To increase the value of our findings and the resulting conclusions we investigated and explored the following moderating variables – *subject*, *experience*, and *gender*.



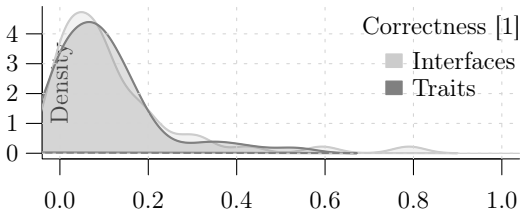
(a) Kernel Density Plot of Overall Comprehend **Duration**



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



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



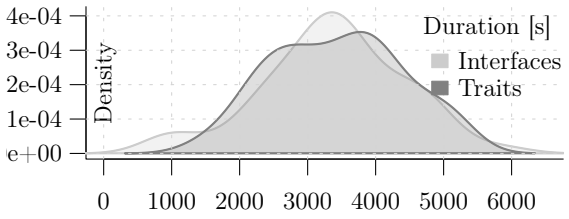
(d) Kernel Density Plot of Task 3 **Correctness**

Figure 7.7: Descriptive Plots per Group of Overall and per Tasks **Correctness**

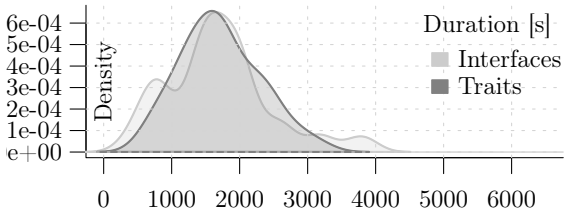
Moderating Variable Subject

For this moderating variable, we are interested to analyze the participants' task-based performance and if such increases or decreases. In order to obtain such results, we first investigated if there is a difference in the processing time. Due to the experiment design (see Section 7.3), we are able to divide the dependent variable *duration* into two parts – comprehend (reading/understanding) and specify (modeling/writing).

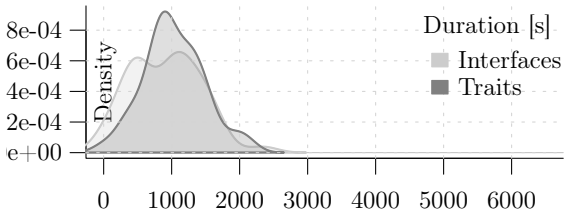
Figure 7.7a depicts the comprehend *duration* for all tasks whereas Figure 7.8a depicts the specify *duration* for all tasks. We can observe from those two kernel density plots that the participants spent more time on the actual specifying process than reading and comprehending the informal specification of the given tasks. For both experimental groups the distribution looks very similar. The comprehend and specify duration can be further analyzed for each task. The comprehend duration has a very



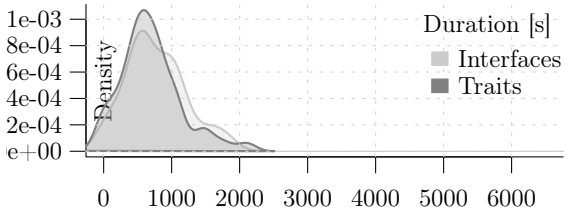
(a) Kernel Density Plot of Overall Specify **Duration**



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



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

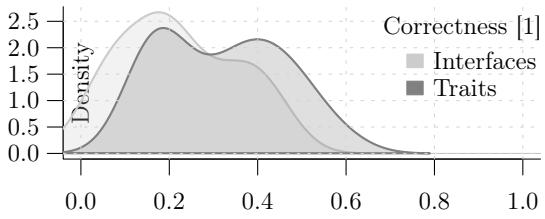


(d) Kernel Density Plot of Task 3 Specify **Duration**

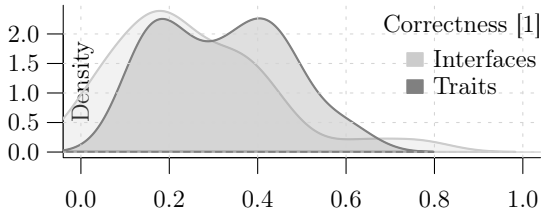
Figure 7.8: Descriptive Plots per Group of Overall and per Tasks **Duration**

similar distribution for all three tasks[26]. For the specify duration we can observe a decreasing effect for the processing time which is visualized for Task 1 at Figure 7.8b, for Task 2 at Figure 7.8c, and for Task 3 at Figure 7.8d. This slight decreasing effect of the specify duration can have two origins. Either the participants experience experimental fatigue [130] or a maturation effect [137] took place. In order to analyze those effects we dissected the dependent variable *correctness* for each task – Task 1 at Figure 7.7b, Task 2 at Figure 7.7c, and Task 3 at Figure 7.7d. We can observe that the traits group performs significantly better for Task 1 and Task 2 compared to the interfaces group. Despite the shorter specify duration (processing time) in Task 2 the correctness and therefore the participants' performance does not degrade at all. But for Task 3 we can detect a complete drop of the participants' performance for both experimental groups which is the result of experimental fatigue.

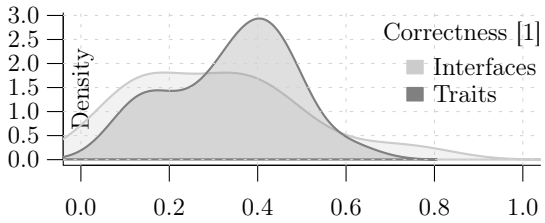
[26]See `appendix.pdf` at [122] for comprehend duration per task plots.



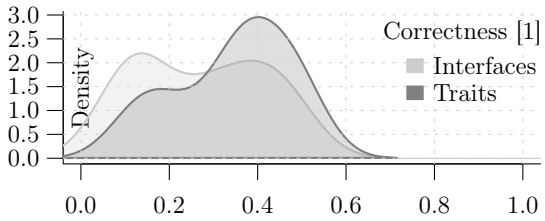
(a) Kernel Density Plot of Less Prog. Exp. **Correctness**



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



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



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

Figure 7.9: Descriptive Plots per Group of **Correctness** by Less/More Experience

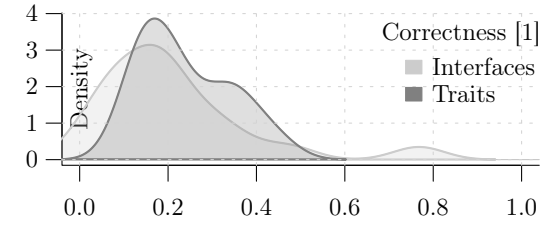
Moderating Variable Experience

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

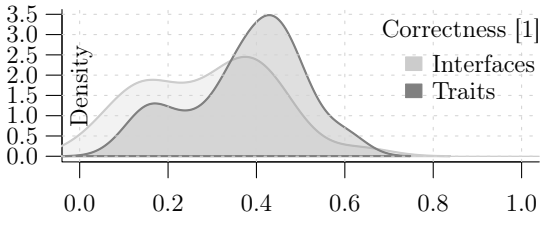
We choose a threshold of 3.25 years in programming experience[27] which results into an exactly equal interfaces to traits sample size ratio for *less* of 28 : 28 and for *more* of 21 : 21. Moreover, we defined that a participant has *less* specifying experience if years $<= 2.5$. From this it follows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

The kernel density plots for programming experience – *less* in Figure 7.9a and *more* in Figure 7.9c – as well as the specifying experience – *less* in Figure 7.9b and *more* in Figure 7.9d – indicate in all distributions the traits group is performing far better than the interfaces group independently of the classification of their experience. Notable to mention here is that the programming and specifying distributions of the *more* experienced participants achieved a high dense correctness value around 0.4. The latter is an indicator why the traits group is performing better in the overall correctness value despite the number of *more* experienced participants is lower than the number of *less* experienced participants.

[27] Abbreviated in Figure 7.9a and Figure 7.9c as "Prog. Exp.".
[28] Abbreviated in Figure 7.9b and Figure 7.9d as "Spec. Exp.".



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



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

Figure 7.10: Descriptive Plots per Group of **Correctness** by Gender

Moderating Variable Gender

With the moderating variable *gender* we will determine an indicator if one of the experimental treatments does perform in terms of correctness better for a certain gender. According to the obtained participants' background information (see Section 7.2) the traits to interfaces sample size ratio for females is 20 : 17 and for males is 29 : 32. Since theses numbers are almost equal within a gender we analyzed for each gender the correctness distributions. Figure 7.10a depicts the kernel density plot for the female correctness whereas Figure 7.10b depicts the kernel density plot for the male correctness. For both gender the traits group performs slightly better than the interfaces group.

Furthermore, we can observe in Figure 7.10a and Figure 7.10b that the participants in this controlled experiment show a clear difference in the performance in terms of correctness depending on the gender. Gren [61] mentions that if there are clear differences in an empirical study based on gender, a proper investigation has to be done to elaborate such effect. By comparing the *gender* results with the data of the *experience* reveals that one possible explanation for the less correct results of the female group can be attributed to lower prior programming experience in the female group compared to the male group.

Threats to Internal Validity

During the experiment, we did not observe any disturbing environmental events or history effects. Due to the total (limited) time of 120 minutes of the experiment, the chances for maturation (carry-over) effects [137] and experimental fatigue [130] were limited. Furthermore, as every participant is only tested once, learning effects can be ruled out. Every participant was able to score the same amount of points and we graded all groups with the same procedures to rule out instrumental bias. Selection bias was limited due to the random assignment of participants to groups. We cannot rule out cross-contamination between the groups as a potential threat to internal validity because the participants are computer science students and share the same social group and interact outside of the research process as well. We have not observed any demoralization or compensatory rivalry. All participants are graded based on their correctness value in the processed survey by gaining points for their enrolled course (but had an opt out option, as explained in Section 7.3).

Threats to External Validity

A possible threat to external validity is that we carried out the experiment with students as participants because this limits the ability to make generalizations. In addition to the types of the participants in this experiment (students as novice software developer or designer), it would be useful to repeat the experiment with broader and more experienced test groups like professionals in different fields ranging

from high-level software design to low-level hardware specifications. Furthermore, the selected experiment tasks are limited to basic software system applications. Due to the usage of the syntax keyword `feature`, we mitigated the risk that the participants are biased by identifying language constructs through known object-oriented abstraction syntax keywords names like `interface` or `trait`. The chosen language construct representations in CASM syntax or their integration into the CASM language might not be representative for potential language constructs and their integration in other ASM languages or other state-based formal languages, and thus our results cannot be generalized to those other languages. We tried to mitigate this threat by only using CASM abstractions that are widely used in other languages, too, and by designing the language constructs as closely as possible to canonical definitions of those abstractions.

Threats to Construct Validity

We focus in this study on the specification effectiveness and efficiency of object-oriented abstractions for an ASM language. The dependent variables *correctness* and *duration* are commonly used to measure the construct **specification effectiveness** and **efficiency**, but other studies use different notations, like Razali et al. [130] which uses *Score (Accuracy)* for **specification effectiveness (correctness)** and *Time Taken* for **specification efficiency (duration)**. Furthermore, other studies analyze both variables under construct names like **comprehensability** (cf. Hoisl et al. [71]) or **understandability** (Czepa et al. [39]). It cannot be ruled out that other constructs would be a better to measure the specification effectiveness and efficiency.

Threats to Content Validity

In this study, we only focus on two object-oriented abstractions, namely interfaces and traits. The specification effectiveness and efficiency is tested for two ASM syntax variations, not commonly existing in today's languages and tools, which use one of the two language constructs (see Section 7.2). Testing more complex scenarios (more complex software system applications and other language constructs) would improve the content validity.

Threats to Conclusion Validity

Due to some missing timestamps for the dependent variable *duration* and unclear written ASM specification solutions for the dependent variable *correctness* we cannot rule out that statistic validity might be affected. Still, those outliers are important measurements because they reflect that for a certain group of the participants the given problem (informal description) to model it through an ASM specification by using a certain language construct are too complex and/or not understood at all. Deleting those would compromise the conclusion validity. To improve the conclusion validity,

we selected robust tests with great statistical power which fits the best explored model assumptions of all statistical tests suitable for the collected data set.

Inferences

Based on the evidence found in this research, a possible use of *Traits* in ASM language designs should provide a good specification effectiveness and efficiency. As *Interfaces* perform significantly worse for the dependent variable **Correctness** than *Traits*, they should be used with more caution. Regarding the dependent variable **Duration**, it seems that for both language constructs the participants need a similar duration to process (read, comprehend, and specify) the tasks and without further studies no generalized claim can be drawn from the gathered results. Taking into account the qualitative measurements, participants using *Traits* without even knowing the language construct specify more efficiently than the *Interfaces* group, which has high familiarity of the language construct (see Section 7.6). Furthermore, the proposed language syntax of the *Traits*-based ASM specification shows very efficient specification performance for expressing structural and behavioral aspects (see Figure 7.5a and Figure 7.5b) which is not the case for experimental group *Interfaces*.

Relevance to Practice

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

The findings in this study are first indicators for specification language designers in practice to choose, specify, and implement new language constructs for existing or newly developed programming or specification languages. This could help to create a more understandable *language syntax* which can be used more effectively and efficiently by a *language user* [83]. Many formalisms, including ASMs, are implemented in different programming and/or specification languages. Our empirical results can help specification language designers to choose one of those languages using the available language constructs in the *language syntax* as a decision criterion (among others) and/or by considering the extensibility of the language options with regard to language constructs. The outcome of this study already has made an impact in the state-based formal method community by introducing a *Traits*-based language construct in the CASM language [116] as described in Chapter 5.

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

7.7 Conclusion

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

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

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

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

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

languages or state-based formal methods.

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

"Hell isn't other people's code. Hell is your own code from 3 years ago." – Jeff Atwood

CHAPTER 8

Eye Tracking Study

Increasingly complex systems require powerful and easy to understand specification languages. In this chapter[1], we report about an eye-tracking experiment we performed during course of the design of the new CASM trait-based language syntax extension. We use our executable specification language based on the ASMs formalism in order to understand how newly introduced language features for formal methods are comprehended by language users. In the course of this study we carefully recruited nine engineers representing a broad range of potential users. For recording eye-gaze behavior we used Pupil Labs eye-tracking headset. An example specification and simple comprehension tasks were used as stimuli. The results of the eye-gaze behavior analysis reveal that the new language feature was understood well, but the new abstractions were frequently confused by participants. The foreknowledge of specific programming concepts is crucial how these abstractions are comprehended. More research is needed to automatically infer this foreknowledge from viewing patterns. This experiment was another follow-up study after the understandability study described in Chapter 6 and the usability study described in Chapter 7.

8.1 Introduction

Because of the increasing complexity of hardware-software systems, interdisciplinary teams are needed to specify and implement them in a robust and especially efficient way. Specification languages like SysML or UML enable communication across disciplines, but fall short when it comes to executable models. Executable specification languages try to fill this gap. A novel formal method based ASM [63] specification language is CASM [117]. Feedback from users, i.e., engineers with various backgrounds in software and/or hardware design and development, is of high importance for the design of such specification languages. As engineers spend more time reading than writing program code, to enable a flat learning curve, and as specification tasks require

[1]The content of this chapter is a revised version of the EMIP'19 paper [141].

many stakeholders (including some non-technical stakeholders) to work together, specification languages require a high degree of comprehensibility.

Eye tracking has been used frequently in computer science [110] to investigate human factors of programs and especially established programming or modeling languages. To our knowledge there is no study using eye gaze behavior as a feedback for designing and improving a specification language yet. In the study described in this chapter, we focus on eye-gaze behavior in order to investigate recently introduced language concepts of the executable specification language CASM.

Our results of the eye-tracking experiment reveal that the syntax of the language extension is well understood, but surprisingly its structural and behavioral elements are confused. Furthermore the evidence becomes apparent that the foreknowledge of specific programming concepts rather than the programming experience is decisive how new programming language concepts are comprehended.

Considering that all participants have never seen the language before and had only a short time to study the main concepts of CASM, the completion time is relatively short. Fixation time and duration are mapped to areas of interest in the presented code during the program comprehension process. Completed by a post-hoc interview, we found that higher fixation duration and fixation counts are not connected to difficulties to understand sections, but indicate a high interest in actively comprehending (learning) the new specification language.

8.2 Background

Specification languages support engineers to capture requirements and specify hardware and software systems in an easy and technology independent way. The ASM theory and its formal methods provide the foundation to make specifications executable. The foundational concepts are: (1) an executable *ASM specification* language which looks similar to pseudo code to express rule-based computations over algebraic functions with arbitrary data structures and type domains; (2) a *ground model* serving as a rigorous form of blueprint and reference model; (3) a step-wise *refinement* of the reference model by instantiating more and more concrete models which uphold the properties of the reference model [26].

Despite its potential existing ASM modeling languages do not gain currency. According to Börger [24] there is the need for better abstractions in existing ASM modeling languages to reach the characteristic of a programming language without focusing on class and inheritance concepts. These abstractions should not come at the cost of increasing complexity but remain comprehensible on a high level.

The CASM aims to bridge this gap by providing these language features. Currently we investigate type abstraction with low implementation overhead on *language engineering* side and high understandability on *language user* side.

The primary intention behind these new type abstractions is to guide programmers

Table 8.1: Participants Experience

Participant	Years	Level	Languages
P1	3	low	Java, JavaScript, Python
P2	10	medium	Java, JavaScript, Python
P4	19	high	Java, JavaScript, Python, VisualBasic, PHP, C++, C#
P5	1	low	Java, C++
P6	15	high	Java, Python, C, Haskell, VHDL
P7	5	medium	Java, JavaScript, C++, C#, PHP
P8	4	medium	Java, Python, C, C#, C++

to use exclusive language features and therefore make specifications more comprehensible. Concretely we introduced new syntax definition elements – `structure`, `feature`, and `implement` – to extend the functionality of structures and their behavior similar to *traits* [134], whereas the syntax is influenced by the Rust [101] programming language[2].

In a previous study [119] (Chapter 6) we compared three different type abstractions *interfaces*, *mixins*, and *traits* in terms of their understandability. The understandability was measured by correctness and response time of the participants performance processing the survey. The results of this paper-and-pencil experiment indicate that type abstractions based on *interfaces* and *traits* were more comprehensible than *mixins*.

This follows another study [121] (Chapter 7) where we compared two different type abstractions *interfaces* and *traits* in terms of their usability. In this experiment the usability was measured by specification effectiveness (correctness) and specification efficiency (duration) of the gathered results from the participants. The study results show that there is a significant difference in terms of specification effectiveness where traits perform better than interfaces.

What is not known so far is how *language users* comprehend these newly introduced structural and behavioral elements and especially how they come to an understanding of these abstractions, i.e. which search patterns are performed while reading and comprehending the specification.

In the described study in this chapter we focus on the *traits* syntax and prepared a simple but realistic specification *TrafficLight* (see Listing 8.1) as a stimulus for the eye-tracking experiment. This sample code extends the basic syntax definition elements from the CASM language[3] [117], namely `function`, `derived` and `rule` by the new `structure`, `feature`, and `implement` definitions.

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

[2]See `https://doc.rust-lang.org/rust-by-example/trait` for Rust language description.
[3]See `https://casm-lang.org/syntax` for CASM syntax description.

can eye-gaze behavior help to identify common search patterns and reveal the effort to comprehend the introduced *traits* syntax?

It's commonly known that engineers learn new language abstractions more efficiently the more language paradigms they know. In this study we refer to programming experience as a combination of the time spent with software programming and also the range of different language paradigms.

In the context of the main research question, we investigate if and to which extent programming experience influences the effectivity to spot and distinguish structural and behavioral abstractions. The related hypothesis is that *language users* with background in Object-Oriented Programming (OOP) languages with common inheritance concepts distinguish less effectively between structural and behavioral elements, while *language users* familiar with various languages paradigms easily spot the structure and behavior elements.

8.3 Experiment

In order to analyze viewing patterns and visual effort during the process of comprehension of the new language features we set up an experiment to measure eye-gaze behavior. We recorded eye movements with a monocular eye-tracking headset from Pupil Labs[4], equipped with a 200Hz eye camera and a world camera with a resolution of 1280x720 pixels. The *pupil capture* software (version 1.10) was used for recording eye movements and the front facing camera as well as the fixation detection and surface mapping. Subsequently eye-gaze data was mapped to a browser window where stimuli were presented. In this experiment each participant viewed a sequence of assignments described below. As the main stimuli a sample CASM specification was chosen. To let participants directly interact with the specification code, we used the browser-based code editor Monaco[5] supplemented with fiducial markers in the corners of the browser window to facilitate the surface mapping. Instructions and comprehension tasks, similar to works of [18] were presented to the participant in addition to the specification code, while eye gaze and world camera recordings were triggered via the web-socket protocol and the Pupil Lab API controlled by the experiment server. Besides eye and world camera raw recordings, we collected pupil positions, pupil diameter, gaze positions, fixation data and the fixations on surface mappings.

Procedure

The procedure of the experiment included (a) the calibration of pupil and eye gaze detection, (b) the presentation of a short introduction to CASM with the basic language features including the newly introduced *traits* concept. (c) When participants hit the *Start* button the recording of eye and world camera was triggered and a brief graphical

[4]See `https://pupil-labs.com` for eye-tracking camera device description.
[5]See `https://microsoft.github.io/monaco-editor` for editor software.

representation of a system followed. Subsequent task assignments were shown on the top of the window. After pressing the *next* button, (d) the source code of the corresponding specification was shown. As the main stimuli of the experiment we have

```
enumeration Phase = { Stop, Go }

structure Light = {
    function phase : -> Phase = { Stop }
}

implement Light = {
    derived phase -> Phase = this.phase

    derived oppositePhase -> = (if phase = Stop then Go else Stop)

    derived isOn  -> Boolean = (phase = Go)

    derived isOff -> Boolean = (phase = Stop)

    rule switch = {
        this.phase := oppositePhase
    }
}

feature TrafficController = {
    derived lights -> [ Light ]

    derived phases -> [ [ Phase ] ]

    derived position -> Integer

    rule nextPosition -> Integer

    rule control = {
        let currentPhase = phases[ position ] in
        let nextPhase = phases[ nextPosition ] in {
            assert( |currentPhase| = |lights| )
            assert( |currentPhase| = |nextPhase| )
            forall i in [ 1 .. |lights| ] do {
                assert( light[i].phase = currentPhase[i] )
                if light[i].phase != nextPhase[i] then
                    light[i].switch
            }
        }
    }
}

structure OneWayStreet = {
    function lights : Integer -> Light = { 1 -> Light(), 2 -> Light() }

    function position : -> Integer = { 1 }
}

implement TrafficController for OneWayStreet = {
    derived lights -> [ Light ] = [ lights( 1 ), lights( 2 ) ]

    derived phases -> [ [ Phase ] ] = [ [ Stop, Stop ], [ Go, Stop ]
                                      , [ Stop, Stop ], [ Stop, Go ] ]

    derived position -> Integer = this.position

    rule nextPosition -> Integer =
        this.position := if position = 1 then 2 else 1
}
```

Listing 8.1: Executable Specification Code (*C*ASM)

chosen the *TrafficLight* example specification (see Listing 8.1) inspired by the traffic light examples from Börger and Raschke [25]. Participants were asked to (e) fill-in the missing statements for each of the following tasks: (1) The central component of this system is structure ... (2) The rule ... defines the main logic of the traffic light signaling. (3) The feature ... is implemented ... times in this specification. (4) The structure ... does not implement a default behavior.

The answers were recorded synchronized with the frame count of the eye gaze recordings. After completion of the experiment (f) a post-hoc interview was conducted to evaluate task difficulty and perceived cognitive load as a ground truth of the measured visual effort [48]. The semi-structured interviews aimed to learn about the participant's professional background and programming experience, concluding with a brief discussion about the experiment and the language itself.

Participants

We recruited nine participants with a broad range of software engineering experience, where low in the table corresponds to intermediate level, medium to advanced and high to professional level correspondingly. Java is the most prevalent programming language among the participants followed JavaScript and Python. Four participants were recruited at a German and five at an Austrian university.

While all have different professional backgrounds and earned or work towards a computer science degree. All participants volunteered to take part in the experiment. It's noteworthy that all participants - except one (P6) - were new to CASM as this study especially aimed to investigate the *first impression* of this specification language.

Table 8.1 provides all gathered participants experience information except for participant P0 and P3, because as described in the following analysis section, the gathered experiment data for those participants had to be excluded from the results due to technical problems during the eye-tracking recording process.

Analysis

Eye-gaze positions and fixation positions mapped to the browser window were calculated using Pupil Labs *capture* software [6].

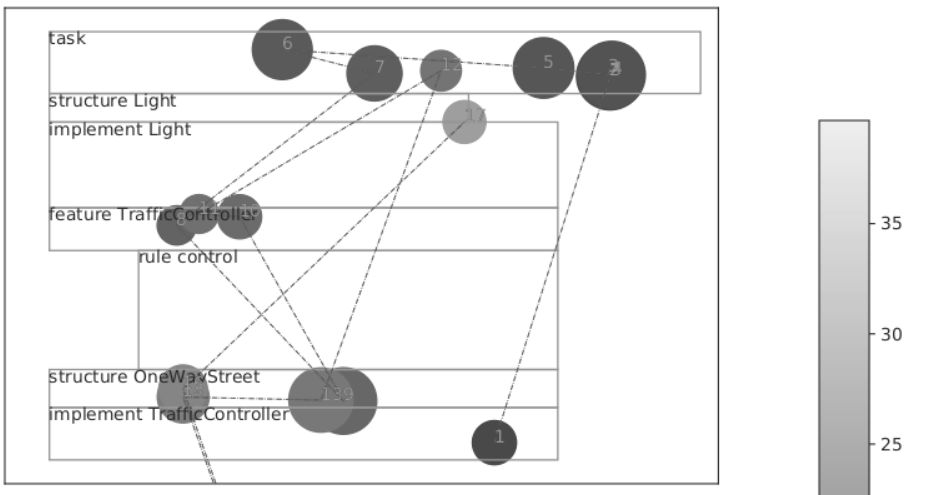
The experiments were conducted in the participant's offices. During two experiments (participants P0 and P3) the front-facing camera of the eye-tracking headset overexposed the white background of the computer screen due to changes in light conditions. These two experiments were excluded from further analysis.

Although the calibration was performed for each participant eye-gaze position data was skewed and had to be recalculated by using an individual offset for each participant. Similar to [13] eye gaze positions were translated into character locations in the web based editor to identify code elements of interest for the user. In total six

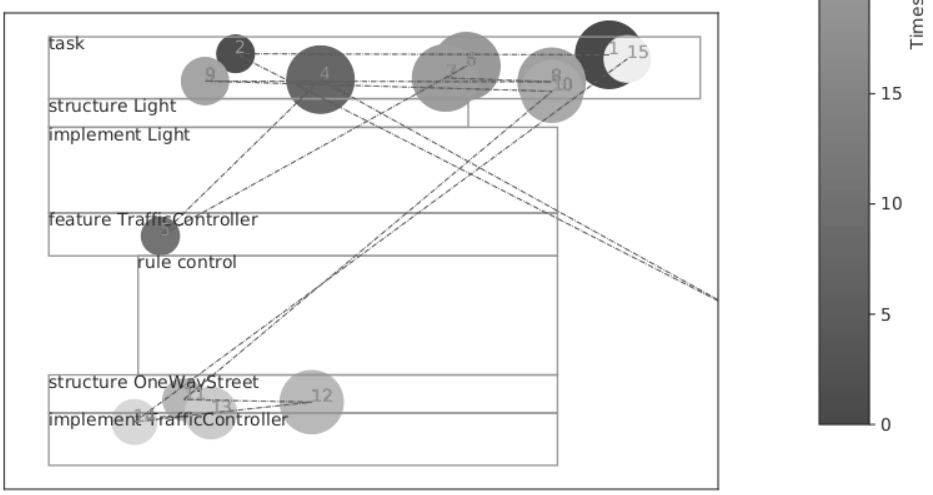
[6] See `https://github.com/pupil-labs` for eye-tracking device and software.

areas representing new language features, i.e. `structure`, `implement`, and `feature` and the task area above the code were selected as main Areas of Interest (AOI). In regard to the research question and to measure the effort to identify and distinguish between structural and behavioral elements of the newly introduced language features, we applied scan path analysis to the eye-gaze data.

As shown in Table 8.3 four participants (P1, P2, P6, and P8) answered three out of four questions correctly, while P1 and P2 did not identify the main component of the system, namely *OneWayStreet* but chose the `feature` *TrafficController* instead. In a further step we selected these experiments to contrast the viewing patterns of incorrectly and correctly answered questions. Figure 8.1 and Figure 8.2 show the indexed fixation positions represented in bubbles whereas the diameter of the circle represents the fixation length. The color of the fixation marker indicates the start time of each fixation. Furthermore the fixations were mapped to the AOIs to facilitate viewing pattern analysis.



(a) Participant 1 (P1)

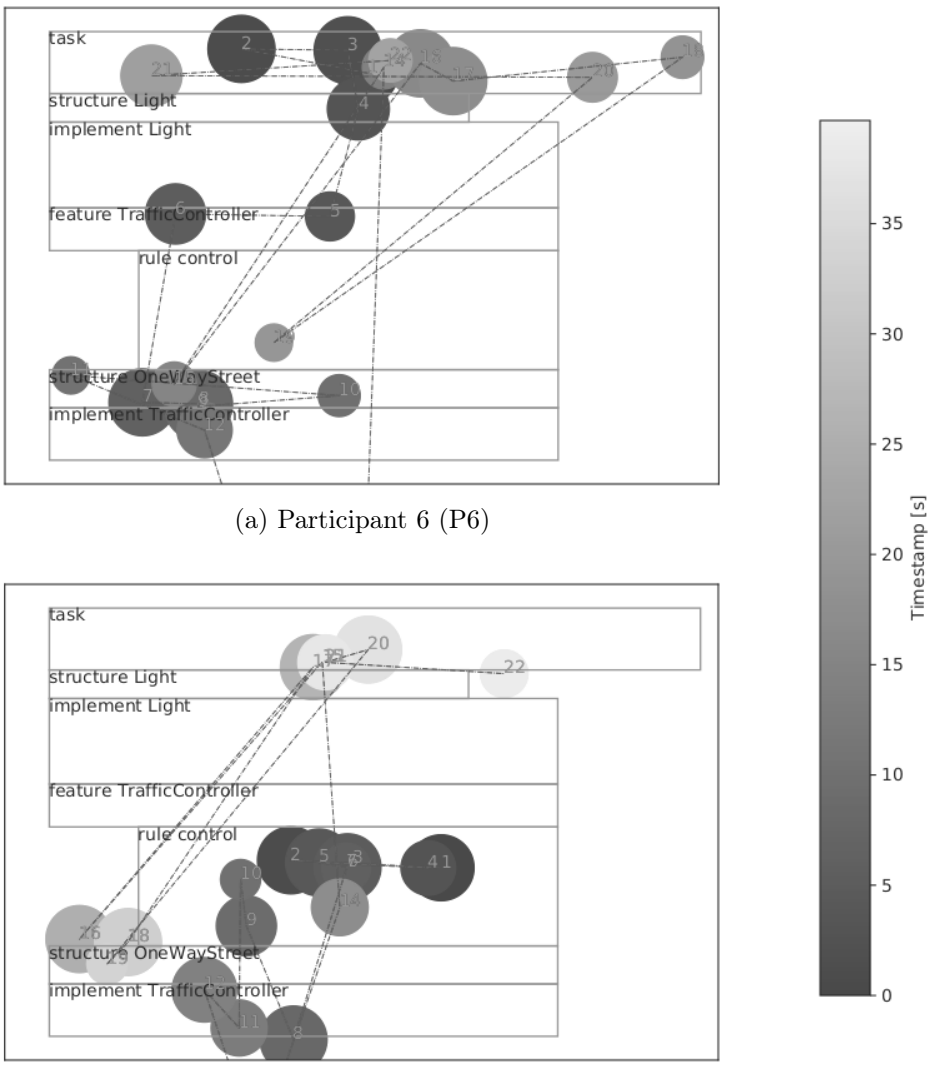


(b) Participant 2 (P2)

Figure 8.1: Fixation Positions for Specification Elements (Incorrect Answers)

The analysis reveals independent of the correctness and completion time a common eye-gaze pattern, i.e. eye-gazes moved between the elements `feature` *TrafficController* and `structure` *OneWayStreet* forth and back, right after reading the assignment. Furthermore the relation of programming experience and the comprehension of the *traits* syntax was investigated. As shown in Table 8.1, Table 8.2, and Table 8.3, task completion time is not correlated to the programming experience level.

While the eye-gaze behavior indicate the experience level, i.e. typical line-by-line reading for novice and source code skimming for professional engineers, viewing patterns explain - to a certain degree - the causal relationship between task correctness and proficiency in language paradigms beyond script languages and OOP. The interviews were analyzed in two ways. The structured data is shown in the Table 8.2 and Table 8.3. The summary of post-hoc open discussion was processed to cluster inferential information and compiled in the results section.



(a) Participant 6 (P6)

(b) Participant 8 (P8)

Figure 8.2: Fixation Positions for Specification Elements (Correct Answers)

Table 8.2: Results of Task 1

Participant	Duration	Correct
P1	22,52s	no
P2	39,58s	no
P4	64,51s	yes
P5	32,14s	no
P6	22,12s	yes
P7	26,00s	no
P8	39,04s	yes

Table 8.3: Results of all Tasks

Participant	Duration	Correctness	Severity
P1	164,57s	75%	5
P2	176,10s	75%	4
P4	237,60s	63%	4
P5	407,42s	13%	5
P6	164,50s	75%	3
P7	190,34s	38%	4
P8	178,91s	75%	3

8.4 Results

Eye-gaze behavior analysis, in general, can not only be used effectively to conclude about the effort in understanding new language designs, but reveals specific issues in comprehension of newly introduced concepts. In particular the results of the scan path and fixation analysis indicate that there is a confusion of `feature` and `structure` supported by the common pattern where participants eye-gaze fixations alternate between the behavioral and the structural elements, exemplified by participant P1's insecurity to choose either *TrafficController* or *OneWayStreet*.

The surprising result of this study is that `feature` is understood as a **structural element** while it's designed as a **behavioral element**.

Language users with experience in OOP languages are commonly used to read the *interface* entirely as structural and behavioral elements can occur at any point in this type of abstraction. In contrast the *traits* syntax clearly distinguishes between structural and behavioral elements. Hence, participants familiar with multiple language paradigms spot these abstractions more effectively, see Figure 8.2a, scan path for participant 6 (P6) for example.

The results of the post-hoc interview complement the findings and act as a ground truth: The `feature` abstraction was perceived as a *new thing*. Furthermore the usage of `assert` and also `enumeration` in this context were somewhat surprising for several participants. The majority of participants found that the preparation time was too short, and suggested to provide printed language instructions or online help to learn about language features.

Although we presented a very simple code editor some participants reported distractions by the editor itself, e.g. the scroll bar. Finally the post-hoc interviews

revealed that participants particular familiar with the *state machines* concept perceived the tasks as highly comprehensible and easy to complete.

8.5 Conclusion

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

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

"Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function." – John Carmack

CHAPTER 9

Conclusion

This PhD thesis describes several achievements for two major research directives regarding the exploration and investigation of ASM-based language engineering – in the field of compiler engineering and language design.

The first part is concerned with the foundation of an ASM-based compiler framework in order to reuse and retarget CASM specified systems to various target domains by following an MDD approach. Therefore, the resulting CASM compiler infrastructure uses a multi-level IR design with a novel ASM-based IR called CASM IR to provide the necessary decoupling from dedicated ASM syntax dialects, and provided the basis for upcoming ASM-based analysis, transformation, and optimization passes. Based on the derived and elaborated CASM IR, the thesis reports about the further exploration of the translation validation capabilities of ASM languages by introducing an improved concolic execution implementation based on in-memory TPTP trace generation for CASM.

The second part of this thesis elaborates the empirical investigation on finding a proper object-oriented language construct and ASM-based syntax extension to introduce an object model into the CASM language and into the ASM method in general. By conducting two controlled experiments and comparing first *interfaces*, *mixins*, and *traits*, it was shown that *interfaces* and *traits* have a similar understandability. A follow-up study has resulted in a significant difference in terms of usability between *interfaces* and *traits*. Therefore, a *trait*-based syntax extension was created and checked for comprehensability by conducting another study through an eye-tracking experiment analyzing eye-gaze pattern behavior and eye fixations of experiment participants. The outcome of all three experiments shaped and created a novel *trait*-based syntax extension for the CASM language.

List of Figures

List of Tables

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.

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

[4] Sven Apel and Don Batory. When to Use Features and Aspects?: A Case Study. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 59–68, New York, NY, USA, 2006. ACM.

[5] Paolo Arcaini, Silvia Bonfanti, Marcel Dausend, Angelo Gargantini, Atif Mashkoor, Alexander Raschke, Elvinia Riccobene, Patrizia Scandurra, and Michael Stegmaier. Unified Syntax for Abstract State Machines. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016*, Lecture Notes in Computer Science 9675, pages 231–236. Springer, 2016.

[6] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. SMT-Based Automatic Proof of ASM Model Refinement. In *Software Engineering and Formal Methods*, pages 253–269, Cham, 2016. Springer International Publishing.

[7] Mark Gordon Arnold. *Verilog Digital Computer Design: Algorithms into Hardware.* Prentice-Hall, Inc., 1998.

[8] Krste Asanović and David A Patterson. Instruction Sets should be Free: The Case for RISC-V. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[9] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[10] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.

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

[12] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving Extensibility Through Product-lines and Domain-Specific Languages: A Case Study. *ACM Trans. Softw. Eng. Methodol.*, 11(2):191–214, April 2002.

[13] Andrew Begel and Hana Vrzakova. Eye Movements in Code Review. In *Proceedings of the Workshop on Eye Movements in Programming*, page 5. ACM, 2018.

[14] Yoav Benjamini and Yosef Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the royal statistical society. Series B (Methodological)*, pages 289–300, 1995.

[15] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. Efficiency of Projectional Editing: A Controlled Experiment. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 763–774, 2016.

[16] Gérard Berry. SCADE: Synchronous Design and Validation of Embedded Control Software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer, 2007.

[17] Sami Beydeda, Matthias Book, Volker Gruhn, et al. *Model-Driven Software Development*, volume 15. Springer, 2005.

[18] Maria Bielikova, Martin Konopka, Jakub Simko, Robert Moro, Jozef Tvarozek, Patrik Hlavac, and Eduard Kuric. Eye-tracking en masse: Group user studies, lab infrastructure, and practices. *Journal of Eye Movement Research*, 11(3), 2018.

[19] Dines Bjørner. The Vienna Development Method (VDM). In *Mathematical Studies of Information Processing*, pages 326–359. Springer, 1979.

[20] Silvia Bonfanti, Marco Carissoni, Angelo Gargantini, and Atif Mashkoor. Asm2C++: A Tool for Code Generation from Abstract State Machines to

Arduino. In *NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 295–301. Springer International Publishing, 2017.

[21] Fred H Borgen and Mark J Seling. Uses of Discriminant Analysis Following MANOVA: Multivariate Statistics for Multivariate Purposes. *Journal of Applied Psychology*, 63(6):689, 1978.

[22] Egon Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *Journal of Universal Computer Science*, 8(1):2–74, 2002.

[23] Egon Börger. The Abstract State Machines Method for High-Level System Design and Analysis. In *Formal Methods: State of the Art and New Directions*, pages 79–116. Springer, 2010.

[24] Egon Börger. Why Programming must be supported by Modeling and how. In *International Symposium on Leveraging Applications of Formal Methods*, pages 89–110. Springer, 2018.

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

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

[27] Gilad Bracha and William Cook. Mixin-Based Inheritance. *ACM Sigplan Notices*, 25(10):303–311, 1990.

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

[29] Josep Carmona, Jordi Cortadella, Mike Kishinevsky, and Alexander Taubin. Elastic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1437–1455, 2009.

[30] Kwang-Ting Cheng and Avinash S Krishnakumar. Automatic Functional Test Generation Using the Extended Finite State Machine Model. In *Design Automation, 1993. 30th Conference on*, pages 86–91. IEEE, 1993.

[31] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model Variables: Cleanly Supporting Abstraction in Design by Contract. *Software: Practice and Experience*, 35(6):583–599, 2005.

[32] Gastón Christen, Alejandro Dobniewski, and Gabriel Wainer. Modeling State-Based DEVS Models in CD++. In *proceedings of MGA, advanced simulation technologies conference*, pages 105–110, 2004.

[33] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996.

[34] Norman Cliff. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological bulletin*, 114(3):494, 1993.

[35] B Copeland. The Manchester Computer: A Revised History Part 2: The Baby Computer. *IEEE Annals of the History of Computing*, 33(1):22–37, 2011.

[36] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of Synchronous Elastic Architectures. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 657–662, July 2006.

[37] Frank P Coyle and Mitchell A Thornton. From UML to HDL: A Model Driven Architectural Approach to Hardware-Software Co-Design. In *Information systems: new generations conference (ISNG)*, volume 1, pages 88–93. Citeseer, 2005.

[38] Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. Traits: An Approach to Multiple-Inheritance Subclassing. In *Proceedings of the SIGOA Conference on Office Information Systems*, pages 1–9, New York, NY, USA, 1982. ACM.

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

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

[41] Luca De Alfaro and Thomas A Henzinger. Interface Theories for Component-Based Design. In *International Workshop on Embedded Software*, pages 148–165. Springer, 2001.

[42] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[43] Giuseppe Del Castillo. The ASM Workbench: A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models. In *Tools*

and Algorithms for the Construction and Analysis of Systems - 7th International Conference, TACAS 2001, Proceedings, pages 578–581. Springer Berlin Heidelberg, 2001.

[44] Jack B Dennis and David P Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. In *ACM SIGARCH Computer Architecture News*, volume 3(4), pages 126–132. ACM, 1975.

[45] Eugene Diirr and J Van Katwijk. VDM++: A Formal Specification Language for Object Oriented Designs. In *Proceedings 6th Annual European Computer Conference, Compeuro*, pages 214–219, 1992.

[46] Olive Jean Dunn. Estimation of the Means of Dependent Variables. *The Annals of Mathematical Statistics*, pages 1095–1111, 1958.

[47] Neil Evans and Michael Butler. A Proposal for Records in Event-B. In *International Symposium on Formal Methods*, pages 221–235. Springer, 2006.

[48] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 26th Conference on Program Comprehension - ICPC '18*, pages 286–296, New York, New York, USA, 2018. ACM Press.

[49] Davide Falessi, Muhammad Ali Babar, Giovanni Cantone, and Philippe Kruchten. Applying Empirical Software Engineering to Software Architecture: Challenges and Lessons Learned. *Empirical Software Engineering*, 15(3):250–276, 2010.

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

[51] Flavio Ferrarotti, Michael Moser, and Josef Pichler. Stepwise Abstraction of High-Level System Specifications from Source Code. *Journal of Computer Languages*, 60:100996, 2020.

[52] Kathleen Fisher and John Reppy. A Typed Calculus of Traits. In *Proceedings of the 11th Workshop on Foundations of Object-oriented Programming*, 2004.

[53] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 171–183, New York, NY, USA, 1998. ACM.

[54] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[55] Robert B France, Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg. Model-Driven Development using UML 2.0: Promises and Pitfalls. *Computer*, 39(2):59–66, 2006.

[56] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, 2014.

[57] Cristian González García, Jordán Pascual Espada, Begoña Cristina Pelayo García Bustelo, and Juan Manuel Cueva Lovelle. Swift vs. Objective-C: A New Programming Language. *IJIMAI*, 3(3):74–81, 2015.

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

[59] David Garlan. 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*, pages 1–24. Springer, 2003.

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

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

[62] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

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

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

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

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

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

[68] Robert Harper, David MacQueen, and Robin Milner. *Standard ML*. Department of Computer Science, University of Edinburgh, 1986.

[69] Alan R Hevner. A Three Cycle View of Design Science Research. *Scandinavian Journal of Information Systems*, 19(2):4, 2007.

[70] Andreas Höfer and Walter F Tichy. Status of Empirical Eesearch in Software Engineering. In *Empirical Software Engineering Issues. Critical Assessment and Future Directions*, pages 10–19. Springer, 2007.

[71] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. 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*, pages 180–189. IEEE, 2014.

[72] James K Huggins and David Van Campenhout. Specification and Verification of Pipelining in the ARM2 RISC Microprocessor. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 3(4):563–580, 1998.

[73] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 209–216. IEEE Press, 2017.

[74] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[75] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[76] Nasser Jazdi. Cyber Physical Systems in the Context of Industry 4.0. In *2014 IEEE international conference on automation, quality and testing, robotics*, pages 1–4. IEEE, 2014.

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

[78] Mark P Jones. The Implementation of the Gofer Functional Programming System. Technical report, Citeseer, 1994.

[79] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically Scheduled High-Level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on FPGAs*, pages 127–136. ACM, 2018.

[80] Natalia Juristo and Ana M Moreno. *Basics of Software Engineering Experimentation*. Springer Science & Business Media, 2013.

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

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

[83] Anneke G. Kleppe. Software Language Engineering: Creating Domain-Specific Languages using Metamodels. *Addisson-Wesley*, 2009.

[84] Philippe B Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[85] William H Kruskal and W Allen Wallis. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[86] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[87] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

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

[89] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization*, pages 75–86. IEEE, 2004.

[90] Roland Lezuo. *Scalable Translation Validation; Tools, Techniques and Framework*. PhD thesis, Vienna University of Technology (TU Wien), 2014. Wien, Dissertation.

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

[92] Roland Lezuo, Ioan Dragan, Gergö Barany, and Andreas Krall. vanHelsing: A Fast Proof Checker for Debuggable Compiler Verification. In *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 167–174. IEEE, 2015.

[93] Roland Lezuo and Andreas Krall. Using the CASM Language for Simulator Synthesis and Model Verification. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, page 6. ACM, 2013.

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

[95] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[96] Roger Lipsett, Erich Marschner, and Moe Shahdad. VHDL - The Language. *IEEE Design & Test of Computers*, 3(2):28–41, 1986.

[97] Barbara Liskov and Stephen Zilles. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, New York, NY, USA, 1974. ACM.

[98] Henry B Mann and Donald R Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The annals of mathematical statistics*, pages 50–60, 1947.

[99] Robert C Martin. Java and C++: A Critical Comparison. *Technical Note, Object Mentor*, 1997.

[100] Robert C Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[101] Nicholas D Matsakis and Felix S Klock II. The Rust Language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[102] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. A Reusable Object-Oriented Approach to Formal Specifications of Programming Languages. In *L'Objet*, pages 273–306, 1998, 4(3).

[103] Marjan Mernik, Xiaoqing Wu, and B Bryant. Object-Oriented Language Specifications: Current Status and Future Trends. In *ECOOP Workshop: Evolution and Reuse of Language Specifications for DSLs (ERLS)*, 2004.

[104] Jason Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179, 2003.

[105] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[106] David A. Moon. Object-Oriented Programming with Flavors. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 1–8, New York, NY, USA, 1986. ACM.

[107] Holly Moore. *MATLAB for Engineers*. Pearson, 5th edition, 2017.

[108] Emerson R. Murphy-Hill, Philip J. Quitslund, and Andrew P. Black. 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, pages 282–291, New York, NY, USA, 2005. ACM.

[109] Steven M Nowick and Montek Singh. Asynchronous Design – Part 1: Overview and Recent Advances. *IEEE Design & Test*, 32(3):5–18, 2015.

[110] Unaizah Obaidellah, Mohammed Al Haek, and Peter C.-H. Cheng. A Survey on the Usage of Eye-Tracking in Computer Programming. *ACM Computing Surveys*, 2018.

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

[112] Flavio Oquendo. π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.

[113] Martin Ouimet and Kristina Lundqvist. The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering. *J. UCS*, 14(12):2007–2033, 2008.

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

[115] Philipp Paulweber, Jürgen Maier, and Jordi Cortadella. Unified (A) Synchronous Circuit Development, 2019. 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC).

[116] Philipp Paulweber, Jakob Moosbrugger, and Uwe Zdun. About the Concolic Execution and Symbolic ASM Function Promotion in CASM. In *International Conference on Rigorous State-Based Methods*, Lecture Notes in Computer Science 12709, pages 112–117. Springer, 2021.

[117] Philipp Paulweber, Emmanuel Pescosta, and Uwe Zdun. CASM-IR: Uniform ASM-Based Intermediate Representation for Model Specification, Execution, and Transformation. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z -*

6th International Conference, ABZ 2018, Lecture Notes in Computer Science 10817, pages 39–54. Springer, 2018.

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

[119] Philipp Paulweber, Georg Simhandl, and Uwe Zdun. On the Understandability of Language Constructs to Structure the State and Behavior in Abstract State Machine Specifications: A Controlled Experiment. *Journal of Systems and Software*, 178:110987, 2021.

[120] Philipp Paulweber, Georg Simhandl, and Uwe Zdun. On the Understandability of Language Constructs to Structure the State and Behavior in Abstract State Machine Specifications: A Controlled Experiment, January 2021. Zenodo, Version 1.0.0, `https://doi.org/10.5281/zenodo.4480316`.

[121] Philipp Paulweber, Georg Simhandl, and Uwe Zdun. Specifying with Interface and Trait Abstractions in Abstract State Machines: A Controlled Experiment. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–29, 2021.

[122] Philipp Paulweber, Georg Simhandl, and Uwe Zdun. Specifying with Interface and Trait Abstractions in Abstract State Machines: A Controlled Experiment, February 2021. Zenodo, Version 1.0.0, `https://doi.org/10.5281/zenodo.4517172`.

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

[124] Kalle A Piirainen and Rafael A Gonzalez. Seeking Constructive Synergy: Design Science and the Constructive Research Approach. In *International conference on design science research in information systems*, pages 59–72. Springer, 2013.

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

[126] Anthony Potts and David H Friedel. *Java Programming Language Handbook*. Coriolis Group Books, 2018.

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

[128] Alexander Raschke and Dominique Méry. Rigorous State-Based Methods. In *Proceedings of 8th International Conference, ABZ 2021, Ulm, Germany, June 9–11, 2021*, page LNCS 12709. Springer, 2021.

[129] Alexander Raschke, Dominique Méry, and Frank Houdek. Rigorous State-Based Methods. In *Proceedings of 7th International Conference, ABZ 2020, Ulm, Germany, May 27–29, 2020*, page LNCS 12071. Springer, 2020.

[130] Rozilawati Razali, Colin F Snook, Michael R Poppleton, Paul W Garratt, and Robert Walters. Experimental Comparison of the Comprehensibility of a UML-based Formal Specification versus a Textual One. In *11th International Conference on Evaluation and Assessment in Software Engineering (EASE) 11*, pages 1–11, 2007.

[131] Marly Roncken, Swetha Mettala Gilla, Hoon Park, Navaneeth Jamadagni, Chris Cowan, and Ivan Sutherland. Naturalized Communication and Testing. In *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 77–84. IEEE, 2015.

[132] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are Students Representatives of Professionals in Software Engineering Experiments? In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 666–676. IEEE, 2015.

[133] Hisashi Sasaki. 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, New York, NY, USA, 1999. ACM.

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

[135] Joachim Schmid. Compiling Abstract State Machines to C++. *Journal of Universal Computer Science*, 7(11):1068–1087, 2001.

[136] Joachim Schmid. Introduction to AsmGofer. *http://www.tydo.de/AsmGofer*, 2001.

[137] SJ Senn. Is the 'Simple Carry-Over' Model useful? *Statistics in Medicine*, 11(6):715–726, 1992.

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

[139] Samuel Sanford Shapiro and Martin B Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4):591–611, 1965.

[140] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[141] Georg Simhandl, Philipp Paulweber, and Uwe Zdun. Design of an Executable Specification Language Using Eye Tracking. In *6th International Workshop on Eye Movements in Programming (EMIP), EMIP'19 at ICSE'19*, May 2019.

[142] Rohit Sinha and Hiren D Patel. Abstract State Machines as an Intermediate Representation for High-Level Synthesis. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[143] Graeme Smith. *The Object-Z Specification Language*, volume 1. Springer Science & Business Media, 2012.

[144] Colin Snook and Rachel Harrison. Practitioners' Views on the Use of Formal Methods: An Industrial Survey by Structured Interview. *Information and Software Technology*, 43(4):275–283, 2001.

[145] Ann E Kelley Sobel and Michael R Clarkson. Formal Methods Ppplication: An Empirical Tale of Software Development. *IEEE Transactions on Software Engineering*, 28(3):308–320, 2002.

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

[147] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[148] Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education India, 2000.

[149] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *Journal of Automated Reasoning*, pages 1–20, 2017.

[150] Stuart Sutherland, Simon Davidmann, and Peter Flake. *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer Science & Business Media, 2006.

[151] Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. Using Students as Subjects – An Empirical Evaluation. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 288–290. ACM, 2008.

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

[153] Yves Vanderperren and Wim Dehaene. From UML/SysML to Matlab/Simulink: Current State and Future Perspectives. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 93–93. European Design and Automation Association, 2006.

[154] Markus Völter. *Generic Tools, Specific Languages*. PhD thesis, Delft University of Technology, June 2014.

[155] Roel Wieringa. Design Science Methodology: Principles and Practice. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 493–494. IEEE, 2010.

[156] Roel Wieringa. Empirical Research Methods for Technology Validation: Scaling up to Practice. *Journal of Systems and Software*, 95:19–31, 2014.

[157] Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

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

[159] Clifford Wolf, Johann Glaser, and Johannes Kepler. YoSys - A Free Verilog Synthesis Suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

Curriculum Vitae

Personal Data

Name	Philipp Paulweber
Date of Birth	December 5th, 1987
Place of Birth	Ehenbichl, Tyrol, Austria
Address	Linke Wienzeile 280 / 185
	1150 Vienna, Austria
Mail	ppaulweber@complang.tuwien.ac.at
Web	`https://ppaulweber.com`
ORCiD	`https://orcid.org/0000-0001-9954-4881`

Education

08/2015 – current	University of Vienna (Universität Wien), Vienna, Austria
	Doctor of Philosophy (PhD), Computer Science
04/2011 – 04/2014	Vienna University of Technology (TU Wien), Vienna, Austria
	Master of Science (MSc), Computer Engineering
03/2008 – 04/2011	Vienna University of Technology (TU Wien), Vienna, Austria
	Bachelor of Science (BSc), Computer Engineering
09/2002 – 06/2007	Mechanical Engineering College (HTL), Jenbach, Tyrol, Austria
	Specializing in Robotics and Automation Engineering
09/1998 – 06/2002	Secondary School (Hauptschule), Ehrwald, Tyrol, Austria
09/1994 – 06/1998	Primary School (Volksschule), Ehrwald, Tyrol, Austria

Experience

07/2020 – current	Vienna University of Technology (TU Wien), Vienna, Austria
	Research Project Assistant, Faculty of Computer Science
	Research Unit of Compilers and Languages
08/2015 – 06/2020	University of Vienna (Universität Wien), Vienna, Austria
	University Assistant (Prae-Doc), Faculty of Computer Science
	Research Group Software Architecture

01/2019 – 06/2020	Fiskaly GmbH, Vienna, Austria Embedded Software Engineer
01/2018 – 12/2018	CodeWerkstatt OG, Vienna, Austria Embedded Software Developer
12/2012 – 07/2015	Time Triggered Technology (TTTech) GmbH, Vienna, Austria Embedded Software Developer
10/2010 – 02/2011	Vienna University of Technology (TU Wien), Vienna, Austria Teaching Assistant (Tutor)
02/2009 – 11/2012	LernQuadrat, Vienna, Austria Private Tutor
07/2008 – 08/2008	Bregenz Festival, Bregenz, Vorarlberg, Austria Audience Service
10/2007 – 12/2007	Funk Fuchs, Sattledt, Upper Austria, Austria Technical Assistant
07/2007 – 09/2007	Plansee AG, Reutte, Tyrol, Austria Production Worker, Crafting Anodes
07/2006 – 08/2006	Plansee AG, Reutte, Tyrol, Austria Production Worker, Crafting Semiconductors
07/2005 – 08/2005	Plansee AG, Reutte, Tyrol, Austria Design Engineer, Constructing Replaceable Cutting Inserts
07/2004 – 08/2004	Plansee AG, Reutte, Tyrol, Austria Technical Maintainer, Repairing Sintering Oven

Publications

Philipp Paulweber, Georg Simhandl, and Uwe Zdun. Specifying with Interface and Trait Abstractions in Abstract State Machines: A Controlled Experiment. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–29, 2021

Philipp Paulweber, Georg Simhandl, and Uwe Zdun. On the Understandability of Language Constructs to Structure the State and Behavior in Abstract State Machine Specifications: A Controlled Experiment. *Journal of Systems and Software*, 178:110987, 2021

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

Philipp Paulweber, Jakob Moosbrugger, and Uwe Zdun. About the Concolic Execution and Symbolic ASM Function Promotion in CASM. In *International Conference on Rigorous State-Based Methods*, Lecture Notes in Computer Science 12709, pages 112–117. Springer, 2021

Philipp Paulweber, Jürgen Maier, and Jordi Cortadella. Unified (A) Synchronous Circuit Development, 2019. 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)

Georg Simhandl, Philipp Paulweber, and Uwe Zdun. Design of an Executable Specification Language Using Eye Tracking. In *6th International Workshop on Eye Movements in Programming (EMIP), EMIP'19 at ICSE'19*, May 2019

Philipp Paulweber, Emmanuel Pescosta, and Uwe Zdun. CASM-IR: Uniform ASM-Based Intermediate Representation for Model Specification, Execution, and Transformation. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018*, Lecture Notes in Computer Science 10817, pages 39–54. Springer, 2018

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

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