Diss. ETH No. 22463
2015

Automated Usable Functional Verification of Object-Oriented Programs

Julian Tschannen

Diss. ETH No. 22463

Automated Usable Functional Verification of Object-Oriented Programs

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
JULIAN MARTIN TSCHANNEN
MSc ETH in Informatik, ETH Zürich, Switzerland

born on
October 27th, 1982

citizen of
Wohlen bei Bern, BE

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Marsha Chechik, co-examiner
Dr. Carlo A. Furia, co-examiner
Dr. Francesco Logozzo, co-examiner
Prof. Dr. Jim Woodcock, co-examiner

2015

Acknowledgments

I thank Bertrand Meyer for giving me the opportunity to do a PhD at ETH Zurich. His support and the programming language he created were the foundation of this work.

Also, this dissertation would not have been possible without the help and supervision of Martin Nordio and Carlo A. Furia. Their expertise and feedback was invaluable for my progress.

A direct impact to this work also comes from Nadia Polikarpova, who designed an awesome framing methodology and was a co-developer of AutoProof during its recent stages, and Christian Estler, who put AutoProof on the web.

Additionally I thank my co-examiners Marsha Chechik, Carlo A. Furia, Francesco Logozzo, and Jim Woodcock for evaluating my dissertation.

My time in this research group was very enjoyable thanks to all the wonderful co-workers: Andrey Rusakov, Benjamin Morandi, Carlo A. Furia, Chandrakana Nandi, Christian Estler, Claudia Günthart, Cristiano Calcagno, Đurica Nikolić, Georgiana Caltais, Jason (Yi) Wei, Jiwon Shin, Marco Piccioni, Marco Trudel, Martin Nordio, Max (Yu) Pei, Michela Pedroni, Mischael Schill, Roman Schmocker, Scott West, Sebastian Nanz, Stephan van Staden.

I also thank Manu Stapf from Eiffel Software, who speedily replied whenever I had a question about EiffelStudio.

During my studies at ETH and before I found wonderful friends with whom I had and continue to have a great time: Antoni Amerena, Antonio Barresi, Till Bay, Nicola Chiapolini, Urs Dönni, Bruno Hauser, Sven Mader, Lukas Naef, Tahmineh Sanamrad, Martin Seiler, Valentin Wüstholz.

Lastly I thank my family, who supported me on my journey all my life, and in particular my wife Maria Gubar (aka *LimKis*), who designed the amazing cover page and illustrations.

Contents

Abstract

Object-oriented programming is pervasive in today's software world; languages based on object-orientation are used from small devices to large projects. Yet, automatic verification of object-oriented programs mostly uses dynamic approaches such as testing. In this thesis we focus on two aspects to improve this situation: (1) supporting full functional verification of object-oriented programs and (2) automated techniques to make verification of object-oriented programs more usable. To attain these goals we have developed several methodologies for automated verification, built an automated verifier, and developed a high-level technique to combine multiple tools in an IDE.

Software developers nowadays can choose from many different tools for analyzing and verifying their code using different techniques both static and dynamic. These tools usually have independent interfaces and may force the programmer to explicitly switch tasks when working with them. We have developed a scoring system to make the process of using several tools in combination more streamlined. Several tools work in the background side by side while an aggregator collects the results and displays a combined score based on the individual results. This frees the developer from invoking tools and unifying the results manually to get the full picture of the state of the program. To demonstrate the technique we built the Verification Assistant combining three tools: a static verifier (AutoProof), a dynamic verifier (AutoTest), and a light-weight code checker (Eiffel Inspector).

With AutoProof we have built a state-of-the-art automated verifier for object-oriented sequential programs with complex functional specifications. AutoProof incorporates two-step verification among other techniques such as semantic collaboration and agent verification. To show AutoProof's performance on verifying object-oriented idioms as well as standard algorithmic problems, we have made an evaluation of AutoProof on a rich collection of benchmark problems from recent verification competitions. The results attest AutoProof's competitiveness among tools in its league on cutting-edge object-oriented verification problems.

To improve feedback of failed verification attempts, we have developed *two-step verification*, a technique that combines implicit specifications, inlining, and loop unrolling. Two-step verification performs two independent verification attempts for each program element: one using standard modular reasoning, and another one after inlining and unrolling; comparing the outcomes of the two steps suggests which elements should be improved.

Both our tools—AutoProof and the Verification Assistant—are available in EVE, the open-source Eiffel Verification Environment. Additionally, AutoProof is available through an on-line interface together with a tutorial, user manual, and a repository with all our solutions to benchmark problems.

Zusammenfassung

Objekt-orientierte Programmierung ist in der heutigen Softwarewelt allgegenwärtig; Sprachen die auf objekt-orientierung aufbauen werden von kleinen Geräten bis zu grossen Projekten verwendet. Trotzdem wird in der Verifikation von objekt-orientierten Programmen hauptsächlich dynamische Ansätze wie zum Beispiel *Testen* angewendet. In dieser Doktorarbeit fokussieren wir uns auf zwei Aspekten um diese Situation zu verbessern: (1) volle funktionale Verifikation von objekt-orientierten Programmen und (2) automatische Techniken, welche die Verifikation von objekt-orientierten Programmen benutzerfreundlicher machen. Um diese Ziele zu erreichen haben wir mehrere Methoden für automatisierte Verifikation entwickelt, einen automatisierten Verifizierer implementiert und eine Technik entwickelt, um mehrere Tools in einer IDE auf hoher Abstraktionsstufe zu kombinieren.

Softwareentwickler können heutzutage von einer Vielzahl von Tools für die Analyse und Verifikation ihre Codes wählen, sowohl Tools die statische als auch dynamische Techniken verwenden. Diese Tools können eigene Benutzeroberflächen anbieten und zwingen so den Programmierer explizit mit diesen Tools zu interagieren. Wir haben ein Punktesystem entwickelt, das die Benutzung mehrere Tools in Kombination vereinfachen soll. Mehrere Tools arbeiten im Hintergrund Seite an Seite, während ein Aggregator die Resultate sammelt und eine kombinierte Punktezahl basierend auf den individuellen Resultaten anzeigt. Dies befreit den Entwickler davon, die Tools einzeln zu verwenden, und die Resultate manuell zu kombinieren um sich ein Bild vom Zustand des Programms zu machen. Um diese Technik zu demonstrieren haben wir den Verification Assistant entwickelt, der drei Tools kombiniert: einen statischen Verifizierer (AutoProof), einen dynamischen Verifizierer (AutoTest), und einen leichtgewichtigen Code-Prüfer (Eiffel Inspector).

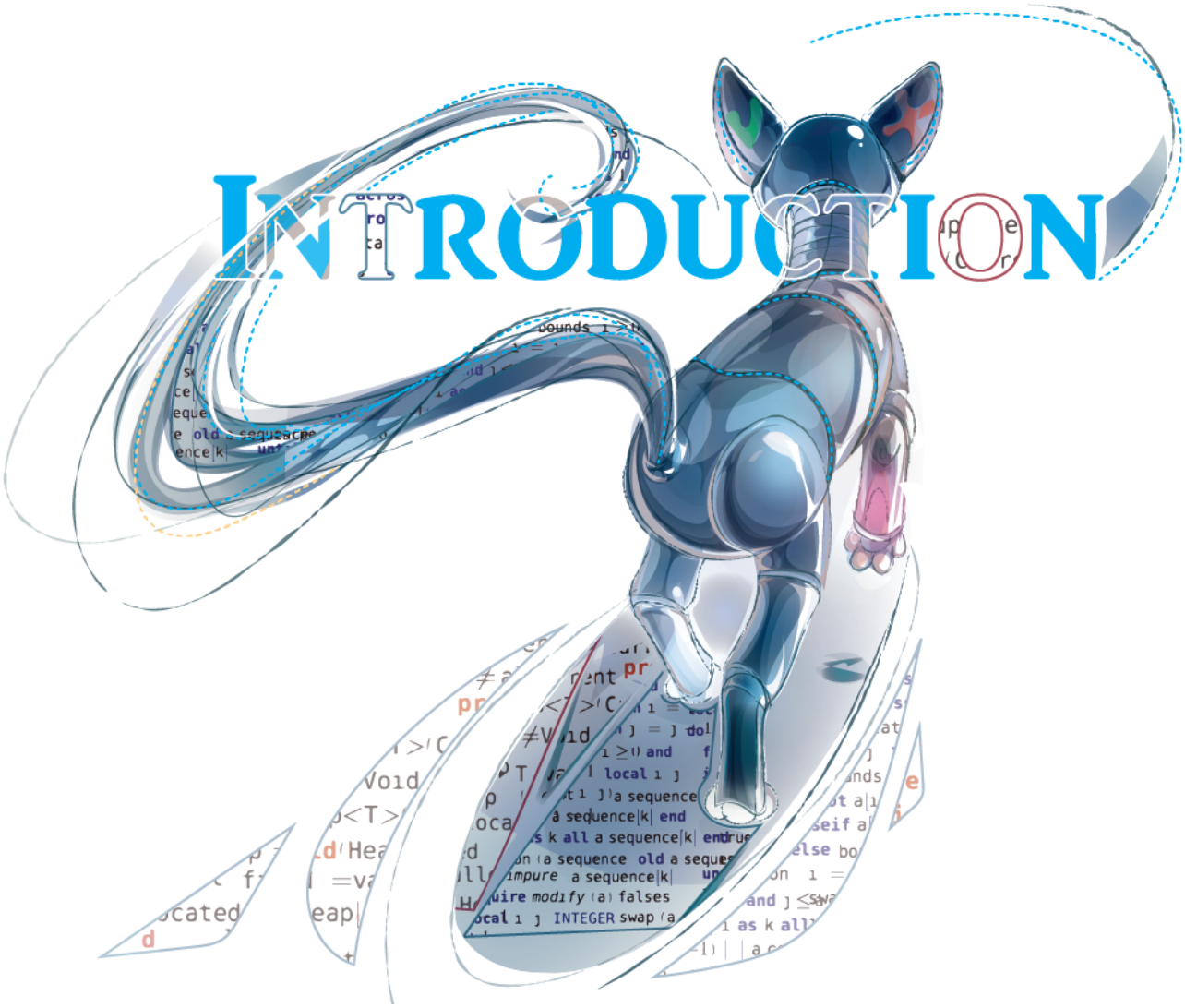
Mit AutoProof haben wir einen modernen auto-aktiven Verifizierer für sequentielle objekt-orientierte Programme mit komplexen funktionalen Spezifikationen entwickelt. AutoProof unterstützt Zwei-Phasen Verifikation und andere Techniken wie Semantische Kollaboration und Verifikation von Agenten. Um die Leistung von AutoProof bei der Verifikation von objekt-orientierten

Idiomen und auch algorithmischen Problemen aufzuzeigen, haben wir eine Evaluation von AutoProof auf einem vielfältigen Benchmark von Problemen kürzlicher Verifikations-Wettbewerbe gemacht. Die Resultate attestieren die Konkurrenzfähigkeit von AutoProof gegenüber anderen Tools in seiner Liga bei aktuellen objekt-orientierten Verifikationsproblemen.

Um das Feedback bei fehlgeschlagenen Verifikationsversuchen zu verbessern haben wir *Zwei-Phasen Verifikation* entwickelt; eine Technik, welche implizite Spezifikationen, Inlining und Loop Unrolling kombiniert. Zwei-Phasen Verifikation führt zwei unabhängige Verifikationsversuche durch für jedes Programmelement: die erste Phase benutzt das modulare Verfahren und die zweite Phase verifiziert nach Inlining und Unrolling. Der Vergleich dieser beiden Phasen weist auf das Programmelement hin, welches zu verbessern ist.

Unsere beiden Tools—AutoProof und der Verification Assistant—sind in EVE integriert, dem quelloffenen Eiffel Verification Environment. Zusätzlich ist AutoProof über eine online Benutzeroberfläche erhältlich, zusammen mit einem Tutorial, einem Benutzerhandbuch und einem Archiv mit unseren Lösungen zu Benchmark-Problemen.

Chapter 1



1.1 Motivation and Goals

Even long-standing skeptics must acknowledge the substantial progress of formal methods in the last decades. Established verification techniques, such as those based on axiomatic semantics or abstract interpretation, have matured from the status of merely interesting scientific ideas to being applicable in

practice to realistic programs and systems. Novel approaches have extended the applicability of these techniques beyond their original scope, providing new angles from which to attack the hardest verification challenges; for example, model checking techniques, initially confined to digital hardware verification, are now applied to software or real-time systems. Tool support has tremendously improved in terms of both reliability and performance, as a result of cutting-edge engineering of every component in the verification tool-chain as well as the increased availability of computing power. Tools have practically demonstrated the impact of general theoretical principles, and they have brought automation into significant parts of the verification process.

In modern object-oriented programming languages, developers write idiomatic object-oriented code based on established design patterns and using language features such as polymorphism, function objects, and exceptions. Large systems are built based on modularity, encapsulation, and information hiding. To provide automated verification of functional correctness of object-oriented programs, developers need to provide machine-readable specifications. To this end, Design by Contract [106] has been adopted by several object-oriented programming languages, either natively [107, 7] or ad-hoc [63, 91]. This paves the way for tools to offer advanced code analysis, automated test generation, and static verification.

Verification of idiomatic object-oriented code faces many challenges created by the flexibility offered to developers, such as framing, aliasing, class invariants, and collaborative object structures. Large systems are built up from smaller components, encouraging modularity also on the level of verification. For developers to use verification tools in practice, tools need to be usable and cover a large part of the programming language. With the advance in verification techniques, the number of tools available to developers is increasing as well. Instead of just using these tools individually, integrating multiple tools in beneficial ways is a challenge as well, that can be addressed on different levels.

This is where this thesis tries to advance the state of the art: our goals are (1) to propose a novel way of integration verification tools, (2) provide tool support for formal verification of an object-oriented language, and (3) improve the usability of verification tools.

1.2 Summary and Main Results

1.2.1 Integration of verification tools

With the increased availability of verification tools, developers have to integrate these tools into the development process. To provide direct integration into the workflow of programmers, tools need to be available as part of the development environment (IDE). Verification tools sometimes offer a standalone graphical user interface (e.g. [51, 67]) or are integrated only loosely into an existing IDE (e.g. [16, 39, 102]). In modern IDEs, multiple such tools are usually available side-by-side [151, 139, 57]. Different approaches have been proposed to leverage the existence of multiple tools by combining them in ways to enhance the individual functionalities, for example by combining multiple analyses [44], making assumptions explicit [34], or by combining static and runtime verification [3]. These integration techniques work on a low level and require the verification tools to be directly compatible in some form, therefore they are not applicable to integrate tools of widely differing techniques.

We propose an approach to combine the output of verification tools that works on a higher level of abstraction than existing techniques. The integrated verification tools work completely independently and report their result to a central controller. Each individual tool's results are summarized as a single correctness score for each unit of code (e.g. routine). In addition, tools report the confidence they have in their own score. The confidence is based on the tool's applicability to the verified code; for example, when a tool verifies a routine that uses floating-point arithmetic, but does not model floating-point numbers according to the underlying machine semantics, the confidence in the result will not be 100%. The controller then combines scores from different tools using weighted averages based on the confidence of each tool and other measures, e.g. visibility of routines. Their aggregated results are displayed to the developer in a single interface, highlighting which areas of the program are in a good or bad shape with respect to the verification effort. The advantage of working at this level of abstraction is that the integration is tool-agnostic: The approach can be used to combine different tools that are completely independent from each other, supports integration of tools that use very different techniques, and simplifies the addition of new tools. We have implemented this methodology in the Verification Assistant as part of the Eiffel Verification Environment (EVE) [62] combining three diverse tools: (1) AutoProof [149], a static verifier; (2) AutoTest [109], an automatic contract-based testing tool; and (3) Eiffel Inspector [157], a light-weight code analysis tool.

1.2.2 Verification of object-oriented programs

Object-oriented programs pose many challenges to verification: accurately modeling heap manipulations in the presence of aliasing, object consistency, or inheritance while still having automated reasoning. Different verification techniques like abstract interpretation [45, 46], model checking [36], or deductive verification [79] can be used to tackle these challenges. While each technique offers different advantages as discussed in Section 1.3, we have chosen to use the approach used by several other verifiers [20, 38, 94, 95, 39] and use Boogie [93] as an intermediate verification language. The use of Boogie as an intermediate verification language allows us to work on a high level of abstraction when reasoning about the semantics of the programming language, Eiffel in our case, and makes sound and precise reasoning about complex properties possible. In addition, using an intermediate verification language allows us to benefit from independent improvements on any element of the tool-chain.

Program verification techniques differ wildly in their degree of automation, ranging from completely *automatic*, where the only input required is the program to be verified, to *interactive* approaches to verification, where the user is ultimately responsible for providing input to the prover on demand. In more recent years a new class of approaches have emerged that try to achieve an intermediate degree of automation between automatic and interactive—hence their designation [97] as the portmanteau *auto-active*[1]. Auto-active tools need no user input during verification, which proceeds autonomously until it succeeds or fails; however, the user is still expected to provide guidance indirectly through *annotations* (such as loop invariants) in the input program. The auto-active approach has the potential of better supporting *incrementality*: proving simple properties would require little annotations and of the simple kinds that novice users may be able to provide; proving complex properties would still be possible but by sustaining a high annotation burden.

With AutoProof we developed an auto-active verifier for the Eiffel programming language. The verifier covers a large part of the Eiffel language and can be used to verify challenging problems. Practical verification of idiomatic object-oriented code is achieved by supporting methodologies such as semantic collaboration [134]—a powerful methodology for framing and class invariants—and verification of function objects [120]. AutoProof has been used to successfully verify a general-purpose container library [133] and to verify client code that uses said library [72]. We have evaluated AutoProof on challenging examples, all of which are available online [13], that

[1] Although *inter-matic* would be as good a name.

attest AutoProof's competitiveness among other tools in its league. We provide an in-depth discussion of a variety of challenges from recent verification competitions in Chapter 4, describing in detail how AutoProof performs on these examples. AutoProof is available for download as part of the Eiffel Verification Environment (EVE) [62] or online on Comcom [9]. The usage is supported by a tutorial [12] and a manual [8] which are present in the Appendices A and B.

1.2.3 Verification methodologies

When developing a verifier for an object-oriented programming language, a crucial design choice are the supported verification methodologies. For example, different approaches to framing (e.g. [136, 84]), dealing with class invariants (e.g. [18, 98, 134]), or dealing with function objects (e.g. [114, 120]) have been proposed, including our own. As part of our work with AutoProof, we have developed new methodologies dealing with specific issues of verifying object-oriented code usable by auto-active verifiers:

- One issue of a failed verification is that an auto-active prover does not indicate if the implementation has a fault or if the specification is too weak or strong. A verification debugger allows developers to manually inspect the state of the verification [76] to gain insight into the underlying problem of the failed verification. We propose a different approach that can in some cases automatically determine if the implementation or specification is to blame for a failed verification. Our approach works by doing an additional verification step after a failed verification while inlining routine calls and unrolling loops. With the information of both verification runs we can try to narrow down the possible reasons for the failed verification. If the second verification attempt is successful, ignoring the specifications of routines and loops, this suggests that the implementation is correct and the specification should be adapted. We call this approach *two-step verification* and use it to improve the feedback after a verification fails in AutoProof.

- In object-oriented programming, subclasses can adapt the specification of routines by weakening the precondition and strengthening the postcondition. However, verifiers tend to simplify this aspect and in general only take static types into consideration, limiting the precision of the verification. We propose to use uninterpreted functions for pre- and postconditions and then linking the actual contracts of a routine to these uninterpreted functions based on the dynamic type of an object. With this approach, implemented in AutoProof, a verifier can use con-

tracts of the dynamic type for verification. This improves the precision of program verifiers when the dynamic type of an object can be inferred at a program location that uses a different static type.

- Verifying code that can throw exceptions introduces particular challenges. When an exception is propagated to the caller, the regular postcondition is in general not established. To address this issue, other verifiers offer two kinds of postconditions wherever necessary [105, 39]: a postcondition for case when the routine exits normally and a postcondition for the exceptional case. Eiffel's peculiar exception mechanism introduces an additional problem on top of that by allowing to restart the execution of a routine in the case of an exception, which is akin to an implicit loop. To verify Eiffel routines that trigger exceptions, we propose a solution that requires additional annotations analogous to loop invariants for the implicit loop introduced by the exception mechanism. We have developed a methodology to annotate Eiffel code that uses exceptions and a translation to an intermediate verification language suitable for implementation in a verifier. The methodology is not implemented in AutoProof, since the Eiffel compiler that AutoProof is built upon implements an old Eiffel exception mechanism, and we have used the Eiffel exception mechanism as proposed in the Eiffel ECMA standard [56] as a basis for the methodology.

1.2.4 Overview of Contributions

This thesis makes the following contributions:

Design of an integrated verification environment [147]

We have developed an approach to integrate verification tools in an IDE, resulting in an *integrated verification environment*. The high-level approach produces correctness scores for each routine and class, based on a combination of correctness scores of individual tools, confidence in the tool's results, and other factors such as importance of routines. We have implemented this method in the Verification Assistant as part of the Eiffel Verification Environment.

Auto-active verification of Eiffel [149, 146]

We have developed AutoProof, an auto-active verifier for Eiffel. The verifier covers a large part of the Eiffel language and can be used to verify challenging problems by supporting semantic collaboration [134] and verification of

function objects [120]. AutoProof has been used to verify a general-purpose container library [133], verify client code of said library, as well as for teaching in a verification course [72]. We discuss in-depth how AutoProof verifies challenging examples and provide an online repository of examples verified with AutoProof. AutoProof is integrated in the Eiffel Verification Environment and available through a web interface.

Methodologies for auto-active verification [148, 150]

We introduce, implement, and evaluate on realistic examples several general methodologies usable by auto-active verifiers.

- *Two-step verification*, a technique to improve the feedback of failed verifications by doing a second verification step that uses inlining and unrolling.
- A methodology to use contracts of the dynamic type whenever the verifier can infer the dynamic type at a program location.
- A methodology to annotate and verify Eiffel code that uses exceptions.

1.3 Related Work

A large amount of research exists in the area of software verification. The work in this thesis falls in a subset outlined by the following criteria:

- Functional verification of object-oriented programs equipped with contracts, including object-oriented idioms like inheritance and collaborative structures.
- Automated verification using an intermediate verification language based on deductive verification.
- Integrating different automated verification tools in an IDE, providing a unified interface.

This section discusses related work for each of these three general areas. Later chapters contain additional related work sections focusing on more specific topics.

Programming Languages With Support for Verification

Verifying correctness of programs written in a general-purpose programming language is a daunting task. To tackle automated verification of functional properties, a specification and the semantics of a programming language needs to be expressible in machine-readable form as well. Some programming languages have been designed with support for specifications, others have support added on top of the language through comments, language extensions, or embedded through existing language mechanisms. The following is a non-exhaustive list of programming languages with support for verification.

Boogie Boogie [93] is both an imperative programming language and a verifier. The language supports first-order logic specifications and the definition of logic theories. The verifier takes a Boogie program and generates verification conditions that can be discharged by SMT solvers such as Z3 [53] or Simplify [54]. Boogie has been used as an intermediate verification language for a number of verifiers, e.g. AutoProof, Dafny [94], or VCC [38]. We use the Boogie language to model and verify the Eiffel language by mapping the object-oriented semantics to a procedural representation.

CodeContracts CodeContracts [63] offers embedded contracts for the .NET platform through a contract library that allows one to express the specifications as part of the program. Developers call the contract library's methods for different types of specifications such as pre- and postconditions, and write the specifications as arguments to these methods. The CodeContracts framework supports both static checking and runtime monitoring of assertions [64] (the latter through a post-build instrumentation step). While CodeContracts offer the full range of specifications—including class invariants and contracts for abstract classes—through method annotations, the CodeContracts Checker [102] focuses on properties verifiable without additional annotations, and does not target complex inter-dependencies idiomatic in some object-oriented patterns. AutoProof takes a different angle, focusing on the verification of complex properties including class invariants and aliasing. This requires users of AutoProof to provide more extensive annotations.

Dafny The Dafny [94] language and verifier were designed to work with specifications based on dynamic frames [84]. Dafny has built-in support for contracts and allows the use of ghost code, making it possible to write expressive specifications. Verification is achieved through a translation to Boogie and Dafny programs can be executed by generating .NET code. Dafny

is well-suited to verify algorithmic problems, but it does not support object-oriented concepts used in mainstream languages, for example inheritance.

Eiffel In our work we have used Eiffel [107], a general-purpose object-oriented programming language that offers embedded contracts and user-defined annotations. Since Eiffel has support for contracts since its creation, Eiffel programmers are familiar with writing contracts [61]. More recent versions of the Eiffel language have added support for loop expressions, which can be leveraged to express bounded quantification in assertions. One of the underlying goals of this thesis is to work towards supporting the static verification of the full Eiffel language.

JML The Java Modeling Language (JML) [91] is a specification language for Java. Contracts are expressed through special comments in the source code. Multiple verifiers exist that work on Java programs annotated with JML, for example KeY [22], ESC/Java2 [40, 32], Krakatoa [66], and OpenJML [39]. These tools focus on simpler properties and do not offer much support for verifying idiomatic object-oriented patterns, for example when collaborative structures are involved. In AutoProof we have implemented a methodology to handle such programs.

SPARK The goal of the SPARK programming language [16] is to provide a language and the tools for high-integrity applications. SPARK is a subset of the Ada language, which adds direct support for contracts in its latest version [2] (previously annotations were given in the form of special comments). To make verification of large systems feasible, SPARK restricts the Ada language to a verifiable subset, for example removing the possibility to manipulate data structures in the heap [29]. This allows the verification of large-scale systems at the cost of reducing programmer flexibility. The latest version of SPARK leverages the Why3 [67] infrastructure for verification.

Spec# Spec# [20, 19] works on an annotation-based dialect of the C# language—in contrast to the newer CodeContracts that follows a library approach—and supports an ownership model which is suitable for hierarchical object structures; as well as visibility-based invariants to specify more complex object relations. Spec# was the forerunner in a new research direction, also followed by AutoProof, that focuses on the complex problems raised by object-oriented structures with sharing, object hierarchies, and collaborative patterns. Collaborative object structures as implemented in practice

require, however, more flexible methodologies [134] not currently available in Spec#, which we developed and implemented in AutoProof.

VCC VCC [38] is a verifier for annotated concurrent C programs. It supports object invariants but with an emphasis on memory safety of low-level concurrent code. VCC programs are verified through Boogie and Z3. Although VCC is very powerful, it does not support object-oriented concepts, which is what we focus on with AutoProof.

Why3 The Why3 [67] language is a dialect of ML—a functional programming language—that allows to write theories as well as contract-equipped programs. Why3 programs can be verified using the Why3 platform and transformed to OCaml programs in a correct-by-construction approach. The Why3 language is used as an intermediate verification language for several languages; for example SPARK, Java (using the Krakatoa verifier [66]), and C (using the Frama-C framework [51]). Why3 is a functional language and focuses on algorithmic problems, whereas we work on object-oriented programs.

Soundness The verification tools mentioned above have different trade-offs with respect to soundness. While a few tools offer sound verification (modulo bugs in the implementation), many tools have sources of unsoundness, either to simplify the analysis in favor of other features such as scalability, or as a result of not accurately supporting some features of the source language.

The two verifiers Boogie [93] and Why3 [67] offer sound verification with respect to their input language (an intermediate language for verification). Soundness is modulo axiomatic theory: both verifiers allow users to manually introduce axioms, which is a source of potential unsoundness if used wrongly.

The CodeContracts [102] checker makes a few unsound assumptions [35], for example with respect to aliasing, trading full functional correctness for speed and large scalability.

ESC/Java2 [40, 32] and OpenJML [39] are both unsound [86, 41]. For example, ESC/Java2 has unsound treatment of machine integers, inherited annotations, or static initializers. Both ESC/Java2 and OpenJML do not support a full-fledged methodology to reason about complex class invariants, such as for collaborative dependencies.

Spec# [20] uses a sound methodology for class invariants, but introduces some unsoundness by treating integer types as mathematical integers and by ignoring exceptional paths.

Our tool, AutoProof, has a sound methodology for complex, collaborative class invariants. AutoProof does not support the full Eiffel language and therefore exhibits unsoundness whenever unsupported code constructs are used (see Appendix B.6 for a detailed list of unsupported language features). It models integers as mathematical integers or as machine integer (with overflow checking) according to a user setting.

The Dafny [94], SPARK [16], and VCC [38] verifiers are sound and fully support their input languages.

Formal Verification Techniques

Formal proofs of programs have been made since before the 1950s [111]. Since then, different automated verification techniques have emerged.

Abstract Interpretation Abstract interpretation [45, 46] uses sound approximation of programs based on abstract domains. Due to the nature of abstraction, false positives might be reported. Techniques exist to improve the precision of abstract interpretation, for example Craig interpolation [49] can be used to guide abstract interpretation away from false alarms [4] by refining invariants produced by abstract interpretation. Abstract interpretation can be used for contract inference [48, 47]. The Boogie verifier uses abstract interpretation to infer loop invariants [17]. The CodeContracts static checker [64, 102] (formerly known as Clousot) uses abstract interpretation to check absence of runtime errors for .NET programs and can verify correctness of .NET programs equipped with contracts. Abstract interpretation targets scalability and speed with little or no manual annotations. It generally focuses on predefined, somewhat restricted properties rather than full functional correctness of object-oriented programs with complex properties. In our work with AutoProof we provide a different trade-off by allowing precise modeling of object-oriented features and complex dependencies, which entails less automation and more annotation burden in exchange for handling arbitrarily complex properties and fine-grained object structures.

Model Checking In model checking [36], the program is modeled as a finite state machine with specifications expressed through temporal logic. The program is checked by exploring the state-space looking for sequences violating the specification; if such a sequence is found, a counterexample is extracted from the model. If the counterexample is spurious, it can be used to refine the abstraction used to obtain the model. This general scheme is known as counterexample-guided abstraction refinement (CEGAR) [37] and used in practice by model checkers such as SLAM [15] or BLAST [23]. Predicate

abstraction [78] can be used to reduce the size of the model by expressing the program through predicates only. In bounded model checking [25, 24], the finite state machine is only explored for a fixed number of steps until a certain bound is reached. Model checkers provide a high degree of automation and can check safety properties without any annotations. We focus on full functional verification of complex properties, which is not the target of model checkers.

Deductive Verification In deductive verification, proof obligations are generated from a program's implementation and specification in a way that the correctness of the proof obligations imply the correctness of the program [79]. The goal is then to discharge the proof obligations using theorem provers. SMT solvers like Simplify [54], CVC3 [21], or Z3 [53] can be used to discharge proof obligations automatically. SMT solvers are sound but in general incomplete, and therefore might report false positives when the verification fails. Interactive proof assistants such as PVS [124], Isabelle [117], or Coq [42] discharge the proof obligations by constructing proofs using a range of proof strategies (also called tactics). When a proof cannot be constructed automatically, the user has to guide the verification by selecting the appropriate proof strategy or introducing intermediate verification steps. The interactive nature allows to verify complex properties only limited by the user himself. AutoProof is a deductive verifier using an SMT solver in the back-end; it is therefore fully automatic but suffers from false positives and might fail to verify a correct program.

IDEs and Frameworks Designed for Verification

In an often parallel effort to increase support for verification in programming languages, development environments and compiler frameworks for specialized and mainstream programming languages have improved support for verification.

Eiffel Verification Environment The tools presented in this thesis have been integrated in the Eiffel Verification Environment (EVE) [62], a research branch of the EiffelStudio IDE [57]. The EVE IDE contains different tools focused on correct software: AutoFix [126], to generate fixes for detected faults; AutoInfer [153], to infer contracts based on program execution; AutoTest [109], an automatic test case generator; and AutoProof [149], an automatic verifier developed as part of this thesis. The Verification Assistant is built on top of AutoTest and AutoProof and integrated in EVE.

Frama-C Frama-C [51] is an extensible platform to analyze and verify C programs. Different tools work with the common specification language ACSL [1]. Frama-C has a consolidation algorithm to combine results from different tools [44]. This allows individual tools to only check a subset of the desired correctness properties and then infer the correctness of the overall program by combining the partial results. Our approach combines verification tools on a higher level, working on the coarser granularity of routines, but allowing to combine widely different techniques. Frama-C provides a stand-alone graphical user interface to evaluate the analysis. We integrated our work in an IDE to offer a single interface for both developing and verifying code.

SPARK Pro SPARK Pro [139] is an integrated development and verification environment for the SPARK programming language [16]. It allows to check a SPARK program for absence of runtime and coding errors. SPARK programs equipped with contracts can be checked for safety and security properties as well as functional correctness. SPARK Pro offers integration of automatic proofs with a framework to write manual unit tests. Tests are only necessary for properties that have not been verified, but they are written manually. In EVE, we focus on integration fully automated verification tools, including an automated verifier and an automated test generator.

UFO The UFO framework [6, 5] is built on top of the LLVM compiler infrastructure [90] and targets the verification of safety properties of sequential C programs. The framework supports model checking techniques based on under-approximation, over-approximation, and combined under- and over-approximation. In contrast, we offer integration of different verification techniques such as static verification and testing.

VisualStudio The VisualStudio [151] IDE features a plug-in architecture supporting multiple programming languages and tools. For the .NET platform, VisualStudio integrates—among other tools—with the automatic test suite generator Pex [143] and offers embedded specifications through CodeContracts [63]. Programs equipped with contracts can be checked statically through the CodeContracts checker [102]. An extension to VisualStudio supporting the Dafny language is available [101]. The integration with Dafny addresses user interface issues existing in other verification IDEs, providing better integration with the back end verifier and debugger through as-you-type feedback and improved messages. In AutoProof we have implemented

two-step verification to improve error messages. Integration with a verification debugger is not yet available in AutoProof.

Why3 Why3 [67] is a programming language and a platform for program verification. The Why3 platform uses multiple back-end verifiers to discharge verification conditions generated from Why3 programs. For this, developers can use both automatic verifiers and interactive proof assistants [27]. The Why3 platform is a potential back-end for AutoProof, but not supported at the moment.

1.4 Outline

This dissertation is organized as follows. Chapter 2 introduces our scoring system for combining verification tools and describes how the Verification Assistant works. The AutoProof tool we have developed and integrated in the Verification Assistant is described in detail in Chapter 3. In Chapter 4 we show an evaluation of the AutoProof tool on challenge problems and its usage in the classroom. Chapter 5 describes new verification methodologies we have developed and implemented in AutoProof. In Chapter 6 we conclude and discuss future work.

The appendix consists of additional resources on AutoProof: Appendix A has a tutorial on AutoProof, covering different aspects of verification with AutoProof. The AutoProof user manual in Appendix B describes in a more systematic way the usage, options, capabilities, and limitations of AutoProof.

Chapter 2

Software developers have different verification tools available nowadays, ranging from dynamic approaches to varying levels of static techniques. In this chapter we will introduce an approach to have a unified interface for a range of analysis and verification tools such as AutoProof to help programmers focus their time and energy where it is most needed.

2.1 Introduction

We present the design of a development environment that seamlessly integrates formal verification with the standard tools offered by programming environments for object-oriented development (editor, compiler, debugger, . . .). The integrated environment is called EVE, built on top of EiffelStudio—the main IDE for Eiffel developers. Section 2.5 describes the engineering of EVE, showing how it takes into account several of the heterogeneous concerns originating from the goal of improving the usability of formal verification, such as user interaction and management of computational resources.

The implementation of EVE, freely available for download [62], continues to evolve as a result of ongoing efforts to integrate more verification techniques and new verification tools. The currently available implementation, illustrated through an example session in Section 2.2, focuses on the integration of two well-known techniques: static verification based on Hoare-style proofs currently implemented in EVE through the AutoProof tool (see Chapter 3), and dynamic analysis based on random testing, implemented through AutoTest [109]. Additionally, EVE integrates a light-weight code analysis tool called Eiffel Inspector [157].

2.2 An Example Session with EVE

Consider the perspective of a user—henceforth called Adam—who is using EVE to develop a collection of data structure implementations. Figure 1 shows portions of Adam's code; the code shown is simplified for presentation purposes, but it reflects real features found in versions of EiffelBase, a standard library used in most Eiffel programs.

The ancestor class `COLLECTION` models generic containers with a well-defined interface including, in addition to other features not shown, routines (methods) `extend` that adds its argument to the collection and `is_equal` which tests for object equality. `extend` is annotated with a precondition (**`require`**) and postcondition (**`ensure`**) which refer to other features of the class (such as `has`) not shown. `extend` is *deferred* (abstract) as it lacks an implementation; `is_equal`'s body, instead, calls a pre-compiled implementation written in C through the **`external`** keyword. This encapsulation mechanism prevents correctness proofs of the routine's implementation (whose source is not accessible); in addition, `COLLECTION` cannot be instantiated and tested because it includes deferred routines. This seems an unfortunate situation for verification, but verification with EVE becomes effective in the two descendants of `COLLECTION` shown in Figure 1: `ARRAY` and `ARRAYED_LIST`.

```
deferred class COLLECTION [G]
  ...
  extendible: BOOLEAN

  extend (v: G)
    -- Add 'v'.
  require
    extendible
    v ≠ Void
  deferred
  ensure has (v) end

  has (v: G): BOOLEAN
    -- Is 'v' in collection?
  deferred
  end

  is_equal (o: COLLECTION [G]):
      BOOLEAN
    -- Is 'o' equal 'Current'?
  require other ≠ Void
  external built_in
  ensure Result = o ~ Current end
end

class ARRAY [G]
inherit COLLECTION [G]
  redefine extendible end
feature
  extendible: BOOLEAN = False
  ...
end

class ARRAYED_LIST [G]
inherit ARRAY [G]
  redefine extendible end
feature
  extendible: BOOLEAN = True

  extend (v: G) do ··· end

  make_default (n: INTEGER)
    -- Allocate list filled
    -- with default values.
  require n ≥ 0
  local l_v: G
  do
    Precursor (n)
    across [1..n] as i loop
      extend (l_v)
    end
  end

  remove_left (c: CURSOR)
    -- Remove item left of 'c'.
  require
    not is_empty
    c ≠ Void and valid (c)
    not c.before and not c.first
  do remove (c.index − 1)
  ensure
    count = old count − 1
    c.index = old c.index − 1
  end
end
```

Figure 1: Classes COLLECTION, ARRAY, and ARRAYED_LIST.

Class ARRAY redefines the attribute `extendible` to **False** because an array is a container of statically-defined size and cannot accommodate new elements at runtime. Correspondingly, the precondition of the inherited feature `extend` becomes unsatisfiable in ARRAY. This way of "deactivating" a routine is inconvenient for automatic testing tools such as AutoTest, which tries, in a vain effort, to generate instances of ARRAY where the precondition of `extend` holds in order to test it. AutoProof, the static proof component of

EVE, comes to the rescue in this case: it easily figures out that the precondition of `extend` is unsatisfiable in `ARRAY` (line 8) in Figure 1), and hence that `extend` is trivially correct and requires no further analysis. Adam checks that `ARRAY.extend` receives a green light and requires no further attention (Figure 2).

Class `ARRAYED_LIST` switches `extendible` to **`True`** and provides a working implementation of `extend` available to clients. When EVE tries to test the class, it quickly discovers a fault in the creation procedure (constructor) `make_default`: after the instruction **`Precursor`** (`n`) calls the creation procedure in the ancestor of `ARRAY`, the loop (**`across`**$\cdots$**`loop`**$\cdots$**`end`**) tries to call `extend` with the local `l_v` as argument; this violates `extend`'s precondition clause `v` $\neq$ **`Void`** because `l_v` is not initialized and hence equals the default value **`Void`** (*null* in Java or C). Adam sees there is something wrong in EVE's report (Figure 2); he expands the description of the error and understands how to fix the bug by adding an instruction **`create`** `l_v` before the loop on line 48.

Item	Score	L	W	Message
ARRAYED_LIST	34	-100	●	1 features failed. 3 features verified.
extendible	78		●	AutoProof: Verified successfully
extend	78		●	AutoProof: Verified successfully
make_default	-100		●	AutoTest: Contract violated
remove_left_cursor	78		●	AutoProof: Verified successfully
ARRAY	100		●	Successfully verified
extend	100		●	AutoProof: Verified successfully
extendible	100		●	AutoProof: Verified successfully

Figure 2: Example report of EVE, showing scores of classes and routines. The third column displays the lowest negative score among the routines of each class.

While Adam is busy fixing the error, testing cannot proceed on the same class. Even if the creation procedure were correct, routine `remove_left` would remain arduous for automated testing techniques because its precondition is relatively complex; a random-based approach to the generation of test cases requires specialized techniques and a long running time to select objects satisfying the clauses in lines 56–58 [154]. EVE circumvents these limitations by running a static proof, which analyzes individual routines and does not need a correct creation procedure. The proof succeeds in establishing that the invocation of `remove` (line 59) is correct and ensures the postcondition of `remove_left`: the routine is correct and no testing is needed (Figure 2).

Later, as soon as the constructor of `ARRAYED_LIST` is fixed, EVE continues its work and exhaustively tests the implementation of `is_equal` finding no postcondition violations. This is not as good as a correctness proof, but it comforts Adam's confidence in the reliability of `is_equal`, and it is the best

result possible for a routine whose implementation can be analyzed only as black-box.

Although it only uses some of EVE's features, this scenario illustrates how EVE can help develop correct applications with little overhead over standard practices:

- EVE is completely automatic and integrated in a full-fledged IDE.
- It supports verification of functional correctness specifications embedded as contracts (pre and postconditions, class invariants, intermediate assertions).
- It transparently manages different verification engines to complement their strengths, supports the full programming language Eiffel, and provides fast feedback to users.
- It only displays such feedback when needed, to encourage focus on the most egregious errors, and to increase the users' confidence in the correctness of an implementation based on the available evidence.

2.3 The Tools of EVE

The integrated verification techniques currently available in EVE and partially illustrated in the preceding example session relies on several tools that are integrated in EVE: AutoProof, AutoTest, and the Eiffel Inspector, which are now presented.

AutoTest

AutoTest [109]—now a standard component of commercial EiffelStudio—is a fully automatic contract-based testing tool. AutoTest generates objects by random calls to creation procedures. Preconditions select valid inputs and postconditions serve as oracles: every test case consists of the execution of a routine on objects satisfying its precondition; if executing the routine violate its postcondition or calls another routine without satisfying its precondition, the routine tested has a fault. AutoTest is fully automatic and does not need any user input. Since contracts are used as test oracles, the tool can find functional correctness bugs. A failing test case provides a concrete error report which is useful for debugging.

Like any dynamic technique based on execution, AutoTest handles every feature of the source language (Eiffel). Among its limitations, instead, is

that random testing can take several hours to find the most subtle faults, and that complex specifications can exacerbate this problem.

AutoProof

AutoProof is an auto-active verifier of full functional correctness and has been described in Chapter 3. AutoProof supports some advanced language constructs such as function objects (agents in Eiffel terminology). Nonetheless, some features of Eiffel—most notably exceptions and floating point arithmetic—are still unsupported and routines using them are not adequately translated to Boogie. The performance of AutoProof depends on the quality of contracts available; accurate contracts improve the modularity of the analysis which can then also verify partial implementations.

Eiffel Inspector

The Eiffel Inspector [157] is a light-weight code analysis tool to find code smells and simple coding errors along the lines of PMD for Java [129] or FxCop for C# [73]. The Eiffel Inspector has an extendible rule system and a large set of implemented rules focusing on code smells like overly complex implementations or simplifiable expressions and also a few coding errors like wrongly used loop iterators. The analysis is in general quick and supports the full Eiffel language.

2.4 Advantages of Being Static and Dynamic

From a user's perspective, EVE's integration of static and dynamic tools can make verification more effective and agile in a variety of scenarios.

- *Modularity and scalability.* Static verification is more modular and scales better to large systems made of several classes. It can also verify routines of deferred (abstract) classes which cannot be tested because they cannot be instantiated. This indirectly improves the performance of testing as well, because the testing effort can focus on routines or classes not proved correct.

  Conversely, whenever testing uncovers a faulty routine, the static tool stops trying to verify that routine. This policy may be broken down to individual clauses: for example, if testing finds a run of `remove_left` (Figure 1) violating the postcondition clause `count` = **`old`** `count` − 1, it may still be worthwhile to try to prove the other clause.

- *Concrete counterexamples.* Dynamic analysis provides concrete reports of errors, which make debugging easier. For example, the following trace documents the error in the creation procedure `make_default` of Figure 1:

```
create {ARRAYED_LIST} l.make_default (1)
-- Inside make_default:
l_v := Void ; Precursor (1) ; extend (l_v) -- l_v is Void
```

- *Partial implementations.* Classes that have a faulty creation procedure or are deferred cannot be instantiated; testing cannot proceed in this case unless the constructor is fixed or an implementation of every routine is available. Static techniques do not incur these limitations: as illustrated in the example of Section 2.2, they can verify individual implemented routines even if others in the same class are deferred.

- *Black-box verification.* Many core libraries rely on routines implemented through calls to low-level external routines (typically, in the Eiffel case, C functions); an example was `is_equal` in Figure 1. Such routines are inaccessible to static analysis but are still testable. The integrated results of static and dynamic analysis on classes with such external routines reinforce the confidence in the correctness of the overall system.

- *Abstract vs. runtime behavior.* Combining static and dynamic analysis can help detect discrepancies between the runtime behavior of a program and its idealized model. Examples are overflows and out-of-memory errors, which are often not accounted for in an abstract specification assuming perfect arithmetic and infinite memory. Consider, for example, a routine that updates the balance of a bank account as a result of a deposit operation:

```
deposit (amount: INTEGER)
  require amount >0
  do balance := balance + amount
  ensure balance >old balance
  end
```

AutoProof with default options models the type `INTEGER` as mathematical integers and verifies that the routine is correct (an option exist to enable overflow checks). AutoTest, however, can still find a bug which occurs when `old balance + amount` is greater than the largest integer value representable and `balance` overflows. It is then a matter of general policy whether one should change the postcondition or the

implementation. In any case, the comparison of the results of static and dynamic analysis clearly highlights the problem and facilitates the design of the best solution.

- *Complex contracts.* Complex contracts considerably slow down automatic testing, both because their runtime evaluation incurs a significant overhead and because random generation takes a long time to build objects that satisfy complex preconditions. Contracts may even in some cases be non-executable because they involve predicates over infinite sets or ghost code; for example, the invariant of a class modeling a hash function requires that the hash code of every possible object (an infinite set) be a non-negative integer. Static techniques can help in all such scenarios: it may be easier to prove the correctness of a routine if the precondition is complex, and hence also stronger; complex postconditions boost modular verification.

These observations highlight the usefulness of treating proofs and tests as complementary and convergent techniques.There is indeed no contradiction; in particular, with the purpose of tests being entirely defined as attempting to make programs fail [108], a useful (that is, failed) test is a proof that the program is not correct. The approach illustrated by EVE is then to combine tools that can prove a program correct (such as AutoProof) and tools that can prove a program incorrect (AutoTest); as soon as a user has written a new program element, the two will start in the background, each with its own specific goal, prove or disprove; in favorable situations, one of them will reach its goal fast, providing the user with a fast response of correctness or incorrectness.

2.5 The Design of an Integrated Verification Environment

The EVE integrated verification environment is built on top of the EiffelStudio IDE and supplements it with functionalities for verification.

2.5.1 Contracts

The choice of Eiffel as programming language ensures that we rely on formal specification elements embedded in the program text as *contracts* (pre and postconditions, class invariants, and other annotations). Since correctness is a relative notion (being dependent on a specification), every verification

activity requires *some form* of specification. Empirical evidence suggests that formal specifications in the form of contracts are a good compromise between the rigor required by formal techniques and the kind of effort that practitioners are able, or willing, to provide [32, 130].

Not all contracts must be written by programmers: the architecture of EVE can accommodate components for *specification inference* to help users write better and stronger contracts. This particular property, however, is not emphasized in the present discussion, which focuses on the integration of static and dynamic analysis assuming some contracts are available.

2.5.2 Automation

A defining characteristic of the tools in EVE is that they are automatic and can do most of the work without any explicit input from the user, assuming the presence of contracts which Eiffel programmers are already used to provide. In order to decouple the machinery of the individual verification tools and to filter out their output, EVE relies on a blackboard architecture [81] shown in Figure 3.

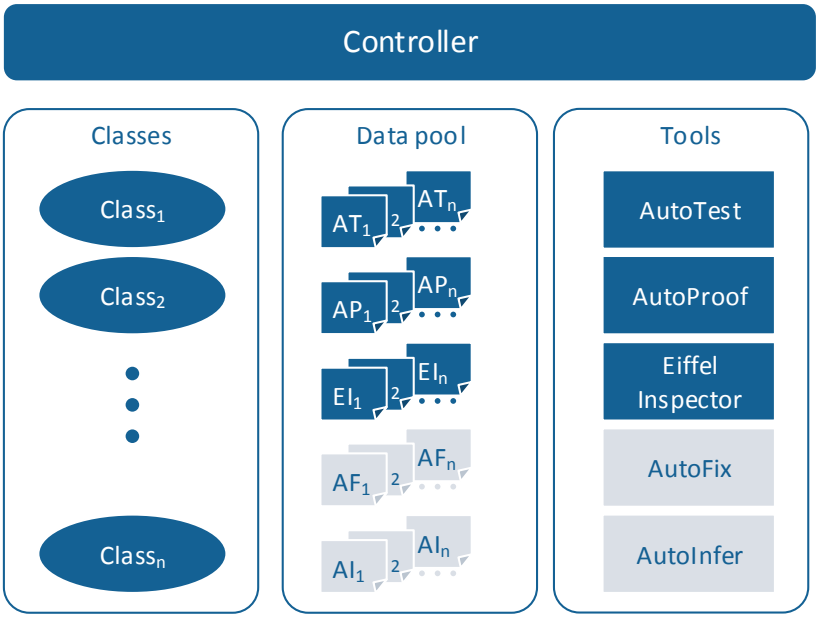


Figure 3: EVE's blackboard architecture (gray boxes currently not implemented).

A *controller* is responsible for triggering the various tools when appropriate, invisibly to the users. The controller bases its decisions on what the user

is currently doing, which resources are available, and the results of previous verification attempts. The latter are collected in a *data pool* where every verification tool stores the results of its runs. Users do not directly read the output of individual tools in the data pool. Instead, the controller summarizes the output data and displays individual tool results only upon explicit user request.

A major design decision of EVE was to make the verification mechanisms as unobtrusive as possible. Users can continue using the IDE and their preferred software development process as before; the verification techniques are an additional benefit, available on demand and compatible with the rest of the IDE's tools. In the same way that type checking adds a new level of help on top of the more elementary mechanisms of syntax error checking, EVE provides reports from proofs and tests on top of the simple verification techniques provided by type checking.

2.5.3 Interaction with the user

Users have currently two ways to control the verification: (1) manually launch the tools or (2) use the fully automatic controller. When using manual mode, the user chooses which tool to run on a specific class. The tool will run in the background and the result will be added to the data pool when available, which will also automatically update the summary display. Automatic mode has only the limited option of being enabled or disabled. Even when enabled, verification never interferes with the more traditional development activities: EVE works in the background on the latest compiled version of the system, and displays a summary of the verification results through an interface similar to that used to signal syntax or type errors in standard IDEs (Figure 2). At any time, the user can browse through the *result list*, which links back to the parts of the program relevant for each message, and decide to revise parts of the implementation or specification according to the suggestions provided by EVE.

Every entry in the result list has a *score*: a quantitative estimate of the correctness or incorrectness of the associated entry, based on the evidence gathered so far by running the various tools. The score varies over the real interval $[-1, 1]$ (In the user interface the scale is, for more readability, -100 to $+100$, with rounding to the closest integer). A positive score indicates that the evidence in favor of correctness prevails, whereas a negative score characterizes evidence against correctness. The absolute value of the score indicates the level confidence: 1 is conclusive evidence of correctness (for example a successful correctness proof), -1 is conclusive evidence of incorrectness (for example a failing test case), and 0 denotes lack of evidence

either way. Figure 2 shows an example of report with scores and stripes colored accordingly. Section 2.6 discusses how the score is computed for the verification tools currently integrated in EVE.

2.5.4 Modularity and granularity

Object-oriented design emphasizes modularity, from which verification can also benefit. While the level of granularity achievable by an integrated verification environment ultimately depends on the level of granularity provided by the tools it integrates, EVE orients verification at the two basic levels of encapsulation provided by the object-oriented model: *classes* and *routines* within a class. EVE associates *correctness scores* with items at both levels. Additional information may be attached to a correctness score, such as the line where a contract violation occurs in a test run, or the abstract domain used in an abstract interpretation analysis. For large systems, it is also useful to have scores for highest levels of abstraction, such as groups of classes or entire libraries, but in the present discussion we limit ourselves to routine and class levels.

The scores from multiple sources of data at the same level are combined with weighted averages, and define the correctness scores at coarser levels. For example, every tool t tries to verify a routine r in class C and reports a correctness score $s_r^C(t) \in [-1, 1]$. The cumulative score for the routine r is then computed as the normalized weighted average:

$$s_r^C = \frac{1}{\sum_{t \in T} w_r^C(t)} \cdot \sum_{t \in T} w_r^C(t) s_r^C(t) \tag{2.1}$$

where $w_r^C(t) \in \mathbb{R}_{\geq 0}$ denotes the weight (or *confidence*) of the tool t on routine r. A similar expression computes the cumulative score s^C for a class C from the scores s_r^C of its routines and their weights w_r^C:

$$s^C = \frac{1}{\sum_{r \in R} w_r^C} \cdot \sum_{r \in R} w_r^C s_r^C \tag{2.2}$$

The weights take various peculiarities into account:

- A tool may not be able to handle certain constructs: its confidence should be scaled accordingly. For example, a tool unable to handle exceptions appropriately has its score reduced whenever it analyzes a routine which may raise exceptions.

- The results of a tool may be critical for the correctness of a certain routine. For example, a quality standard may require that every public

routine be tested for at least one hour without hitting a bug; correspondingly, the weight $w_r^C(t)$ for public routines r would be high for testing tools and low (possibly even zero) for every other tool.

- The correctness of a routine may be critical for a class; then the routine score should have a higher weight in determining the class cumulative score.

- More generally, the weight may reflect suitable *metrics* that estimate the criticality of a routine according to factors such as the complexity of its implementation or specification, whether it is part of the interface of public, and the number of references to it in clients or within the containing class.

- Similar metrics are applicable at other levels of granularity, for example to weigh the criticality of a class within the system.

EVE provides default values for all the weights (Section 2.6), but users can override them to take relevant domain knowledge into account.

2.5.5 Extensibility

The architecture of EVE is *extensible* to include more tools of heterogeneous nature. The user interface will stay the same, with the blackboard controller being responsible for managing the tools optimally and only reporting the results through the summary scores described above.

The architecture can also accommodate tools that, while not targeted to verification in a strict sense, enhance the user experience. For example, tools for assertion inference—such as AutoInfer [153] developed by other members of my group—can complement user-provided contracts and improve the performance of approaches that depend on contracts. The controller can activate assertion inference when the verification machinery performs poorly and when metrics suggest that the code is lacking sufficient specifications. The assertion inference tools themselves may sometimes re-use the results of other tools; for example AutoInfer relies on the test cases generated by AutoTest. Finally, EVE can show the inferred assertions in the form of suggestions, in connection with the results of other verification activities. For example, it could display an inferred loop invariant with the report of a failed proof attempt, and suggest that the invariant can make the correctness proof succeed if added to the specification. The current implementation of EVE does not integrate such suggestions mechanisms yet.

2.6 Correctness Scores for EVE

Equation 2.1 on page 25 gives the correctness score for a routine r of class C; now, consider a set of tools $T = \{p, t, e\}$, where p denotes AutoProof, t denotes AutoTest, and e denotes Eiffel Inspector.

2.6.1 General principles for scores

We noted earlier that an interesting test, that is to say a failed test, is a proof of incorrectness. This is of course another form of Dijkstra's famous observation about testing—but restated as an argument *for* tests rather than a criticism of the notion of testing. This observation has two direct consequences on the principles for computing correctness scores.

First, it is relatively straightforward to assign scores to the result of a testing tool when it reports errors: assign a score of -1, denoting certain incorrectness, to every routine where testing found an error. In certain special circumstances, the score might be differentiated according to the severity of the fault; for example a bug that occurs only if the program runs for several hours may be less critical than one that occurs earlier, if the system is such that it is reset every 45 minutes. In most circumstances, however, it is better to include such domain information in the specification itself and to treat every reported fault as a routine error. Then, different routines may still receive a different weight in the computation of the score of a class (Equation 2.2 on page 25)—for example, a higher weight to public routines with many clients.

The second consequence is that it is harder to assign a positive score sensibly to routines passing tests without errors. It is customary to assume that many successful tests increase the confidence of correctness; hence, this could determine a positive correctness score, which increases with the number of tests passed, the diversity of input values selected, or the coverage achieved according to some coverage criteria such as branch or instruction coverage. In any case, the positive score should be normalized so that it never exceeds an upper limit strictly less than 1, which denotes certain correctness and is hence unattainable by testing.

For verification tools that are sound, a successful proof should generally give a score of 1. Certain aspects of the runtime behavior, such as arithmetic and memory overflows as discussed above, may still leak in some unsoundness if the static verifier does not model them explicitly; in such cases the score for a successful proof may be scaled down in routines with a specification that depends on such aspects.

Which score to assign to a static verifier reporting a failed proof attempt

depends on the technique's associated guarantee of *completeness*. For a complete tool, a failed proof denotes a certain fault, hence a score of -1. If the tool is incomplete, a failed proof simply means "I don't know"; whether and how this should affect the score depends on the details of the technique. For example, partial proofs may still increase the evidence for correctness and yield a positive score.

2.6.2 Score and weight for AutoTest

If AutoTest reports a fault in a routine r of class C, the correctness score $s_r^C(t)$ becomes -1. This score receives a high weight $w_r^C(t) = 100$ by default; the user can adjust this value to reflect specific knowledge about the criticality of certain routines over others with respect to testing.

When AutoTest tests a routine r of class C without uncovering any fault, the score $s_r^C(t)$ increases proportionally to the length of the testing session and the number of test cases executed, but with an upper limit of 0.9. With the default settings, this maximum is reached after 24 hours of testing and 10^4 test cases executed without revealing any error in r. Users can change these parameters; the default settings try to reflect the specificities of random testing shown in repeated experiments [155]. We decided against using specific coverage criteria such as branch coverage in the calculation of the routine score, as the experiments suggest that for example the correlation between branch coverage and the number of uncovered faults is weak.

2.6.3 Score and weight for AutoProof

AutoProof implements a sound but incomplete proof technique. The score $s_r^C(p)$ for a routine r of class C is set accordingly:

- a successful proof yields a score of 1;
- an out-of-memory error or a timeout are inconclusive and yield a 0;
- a failed proof with abstract trace may be a faint indication of incorrectness: the abstract trace may not correspond to any concrete trace (showing an actual fault), but it often suggests that a proof might be possible with more accurate assertions. The score is then -0.3 to reflect this heuristic observation.

The weight $w_r^C(p)$ takes into account the few language features that are currently unsupported: if r's body contains a feature that is not supported (e.g. exceptions), $w_r^C(p)$ is conservatively set to zero, if r's body contains a

feature that is partially supported (e.g. floating point arithmetic), $w_r^C(p)$ is set to a value between 0 and 1. In all other cases, $w_r^C(p)$ is 1 by default.

2.6.4 Score and weight for Eiffel Inspector

The Eiffel Inspector, unlike AutoTest and AutoProof, does not focus on functional correctness. It has only very few rules that reveal a definite coding error and most of the rules indicate bad code quality or even just adherence to coding guidelines. While rule violations in general indicate bad code quality they lend only little evidence for incorrectness and the absence of rule violations has no value for correctness at all. We account for that by setting the score $s_r^C(e)$ and weight $w_r^C(e)$ accordingly.

The score given by Eiffel Inspector reflects the kind of rules violated. Each rule is categorized as either a *hint*, a *warning* or an *error*. We ignore the hint category entirely, as this is only indicative of coding style. If Eiffel Inspector finds a violation of an error rule the score is set to -1 and if only warning rules are violated we give a negative score up to a maximum of -0.3 depending on the number of rule violations to reflect the intuition that complex code has a higher probability of errors. The increase to the negative score is linear in the number of rule violations found and capped at the maximum value of -0.3.

The weight also depends on the type of the violated rules. We set the weight to 100 if one of the few error rules is violated, similar to the case where AutoTest finds an error in the program. When only warning rules are violated we set the weight to 1 and when no rules are violated the weight becomes 0. Therefore, the Eiffel Inspector does not contribute to the overall score if it does not find any rule violations.

2.6.5 Routine weights

Equation 2.2 (page 25) combines the scores s_r^C of every routine r of class C with weights w_r^C to determine the cumulative score of C. The weights w_r^C should quantify the relevance of routine r for the correctness of class C. Different metrics could be used to automatically infer the importance of a routine:

- Routines usable by other classes might have a higher impact on the perceived correctness of a class.

- Routines with higher complexity can be considered more important. A high complexity indicates that the routine is performing an important

operation, therefore its correctness should be more important for the correctness of the class.

- Faults in a routine with many clients have potentially higher impact. When a routine has many clients, its importance could be increased accordingly.

- Using run-time profiling information, the relative time spent executing a routine or the relative number of calls to a routine could be used to weigh the importance of routines. The higher such numbers—execution time or calls to a routine—the higher the importance of a routine.

Despite the possibility to infer the routine importance automatically, it is the developer who has a better understanding of the overall system design and can decide best which routines are more important.

EVE supports a simple way give developers control over the importance setting of routines: every routine has an optional *importance* flag which takes the values *low*, *normal* (the default), and *high*. w_r^C is then

$$w_r^C \;=\; v_r^C \cdot i_r^C$$

The visibility of r determines v_r^C, which is 2 if r is public and 1 otherwise. The importance of r determines i_r^C, which is 2 if r has high importance, 1 if it has normal importance, and 1/2 if it has low importance.

2.7 Usage Scenarios

How serviceable is EVE's score which combines the results of different verification tools, as opposed to considering the tools' outputs individually? This section outlines a few straightforward scenarios that compare the output given by AutoProof or AutoTest in isolation against EVE's combined output; they show the greater confidence supplied by EVE, and the straightforward interpretability of its output. The example models attributes of an individual with a class `PERSON`. Table 4 lists 8 routines of the class to be verified; for each routine, the table reports the score and weight of AutoProof and AutoTest within EVE, and the corresponding combined score.

Routines `set_age` and `set_weight` demonstrate a favorable scenario, where AutoProof and AutoTest can provide strong positive evidence indicating correctness and Eiffel Inspector finds no rule violations. The overall score is, correspondingly, quite high, but it still falls short of the maximum because testing can never prove the absence of errors with 100% confidence.

Item	Tool	Result	Weight	Score
set_age	AutoProof	Verified successfully	1.0	1.00
	AutoTest	No errors found	1.0	0.90
	Eiffel Inspector	No violations found	0.0	0.00
	Routine score			**0.95**
set_weight	AutoProof	Verified successfully	1.0	1.00
	AutoTest	No errors found	1.0	0.90
	Eiffel Inspector	No violations found	0.0	0.00
	Routine score			**0.95**
set_height	AutoProof	Postcondition violation	1.0	-0.30
	AutoTest	Postcondition violation	100.0	-1.00
	Eiffel Inspector	Error: self-assignment	100.0	-1.00
	Routine score			**-1.00**
set_name	AutoProof	Proof failed	0.5	-0.30
	AutoTest	No errors found	1.0	0.90
	Eiffel Inspector	No violations found	0.0	0.00
	Routine score			**0.50**
increase_age	AutoProof	Verified successfully	0.5	1.00
	AutoTest	Overflow detected	100.0	-1.00
	Eiffel Inspector	No violations found	0.0	0.00
	Routine score			**-0.99**
age_difference	AutoProof	Verified successfully	0.5	1.00
	AutoTest	No errors found	1.0	0.90
	Eiffel Inspector	Warning: local not read	1.0	-0.10
	Routine score			**0.60**
print_id_card	AutoProof	Inapplicable	0.0	0.00
	AutoTest	No errors found	1.0	0.90
	Eiffel Inspector	No violations found	0.0	0.00
	Routine score			**0.90**
apply_command	AutoProof	Verified successfully	1.0	1.00
	AutoTest	Inapplicable	0.0	0.00
	Eiffel Inspector	No violations found	0.0	0.00
	Routine score			**1.00**
PERSON	**Class score**			**0.41**

Table 4: Individual and combined results for class PERSON.

Routine set_height shows the opposite scenario, where all three tools report a failure. The routine has an incorrect implementation which triggers one of the few *error* rules of Eiffel Inspector. AutoProof and AutoTest both report a postcondition violation. The two tools AutoTest and Eiffel Inspector have very high confidence when they find errors, thus they both use a very high weight.

Routine set_name relies on the object comparison semantics, which AutoProof overapproximates. In this case, a failed proof does not necessarily indicate an error in the routine, hence it only accounts for a mildly negative score. When AutoTest does not find any error after thorough testing, the combined score becomes visibly positive, while still leaving a margin of uncertainty given the lack of conclusive evidence either way.

Routine increase_age includes integer arithmetic, which might produce overflow. AutoProof can verify the routine, but EVE is aware that overflow

checking is disabled and the proof scheme models integers as mathematical integers, hence it weights down the value of the successful proof because the abstraction may overlook overflow errors. Indeed, AutoTest reveals an overflow when executing the routine with the maximum integer value. The combined score indicates that there is an error, which AutoTest discovered beyond the limitations of AutoProof. Another routine `age_difference` also uses integer arithmetic but it is correct. EVE still scales down AutoProof's score accordingly; in this case, however, AutoTest does not find any error, hence the overall score grows high: the uncertainties of the two tools compensate each other and the cumulative score indicates confidence. Additionally, Eiffel Inspector reports a warning for this routine due to some leftover debugging code which slightly reduces the overall score of the routine.

Routines `print_id_card` and `apply_command` demonstrate how EVE's combination of tools expands the applicability of verification: `print_id_card` uses string manipulations and console output, unsupported by AutoProof, whereas `apply_command` uses agents, unsupported by AutoTest. EVE relies entirely on the only applicable tool in each case.

The overall class score (last line of Table 4) uses a uniform weight for the routines; the score concisely indicates that considerable effort has been successfully invested in the class's verification, but some non-trivial issues are open.

2.8 Related Work

A few authoritative researchers have pointed out the potential of combining static and dynamic techniques [135, 58, 142] to make verification more usable.

Some of the aforementioned testing tools [75, 138, 30] already leverage lightweight static analyses to boost the performance of automated testing. Pex [143] is another scalable automatic testing framework, which relies more heavily on static methods: it exploits a variant of dynamic symbolic execution where an automated theorem prover (Z3) analyzes the symbolic executions to improve code coverage. Pex uses parameterized unit tests [144] as specifications. This makes it possible to test fairly sophisticated properties, but it also requires users to produce specifications in this customized form; contract specifications, however, seem more palatable to practitioners [31].

DASH [135] combines static and dynamic verification with an approach extending the software model-checking paradigm [23]: DASH's algorithm to generate exhaustive tests maintains a sound abstraction of the program, which can be used to construct automatically a correctness argument.

The few recent attempts at combining static and dynamic techniques

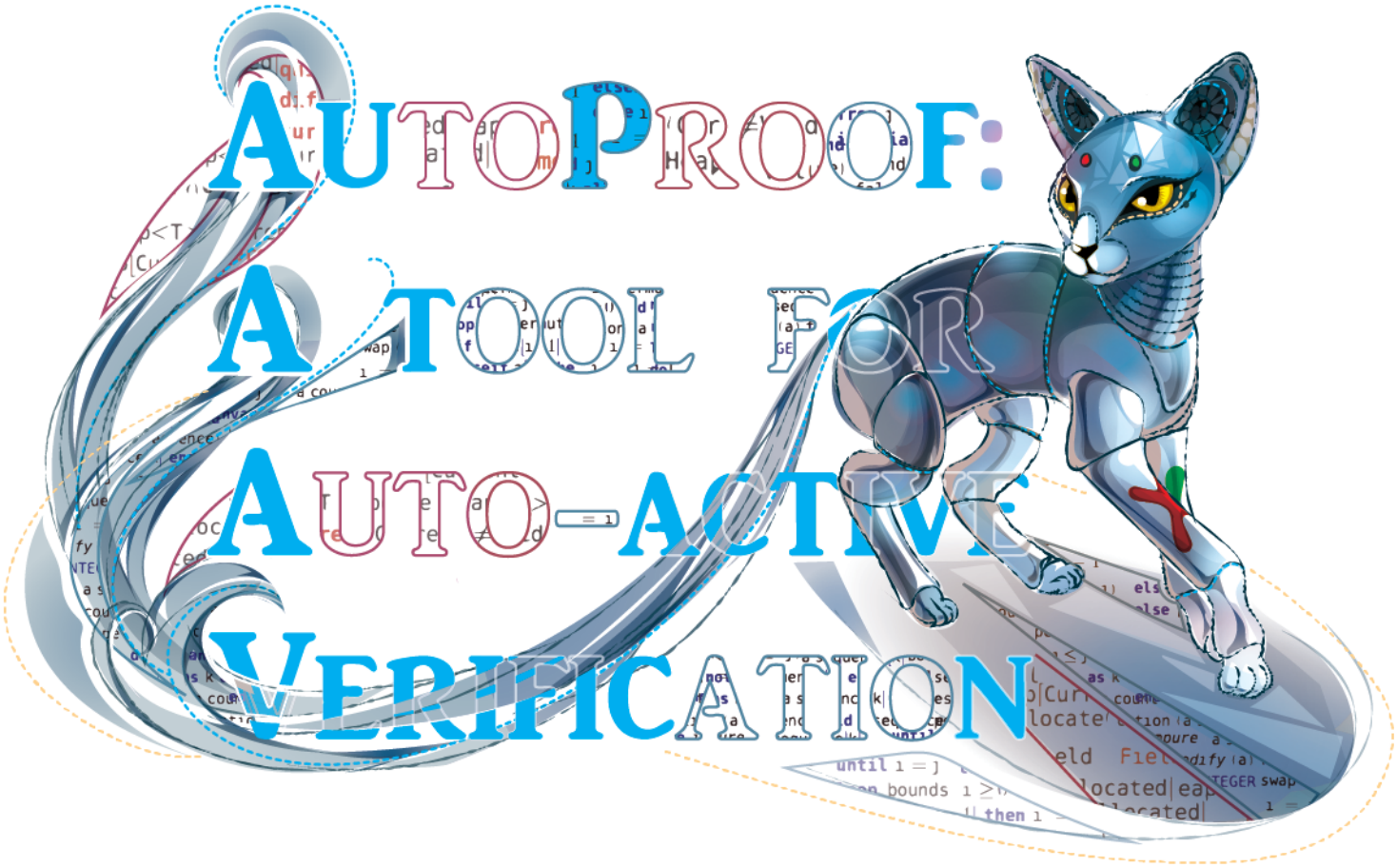
tend to be specific conservative extensions of basic methods; the approach described here tries integration at a higher level to avail the complementarity of static and dynamic techniques to a larger extent.

The SPARK Pro IDE [139] provides analysis and verification tools for the SPARK programming language [16]. The tools are integrated in the user interface and provide feedback through errors and warnings for different types of analyses and—given contract equipped program—verify the functional correctness of the program.

Frama-C [51, 44] is a tool-set to statically analyze C code. In addition to providing a plug-in based system for different types of analyses, Frama-C can be used to verify properties expressed in the ACSL specification language [1].

SIDE [103], a Semantic IDE, leverages the CodeContracts checker [64] to analyze programs in real time. The analysis can be used to verify absences of runtime errors, suggest fixes for errors, and infer contracts in the form of pre- and postconditions.

Chapter 3



This chapter describes AutoProof, our auto-active verifier for functional properties of object-oriented programs. In its latest development state, AutoProof offers a unique combination of features that make it a powerful tool in its category and a significant contribution to the state of the art. AutoProof targets a real complex object-oriented programming language (Eiffel)—as opposed to more abstract languages designed specifically for verification. It supports most language constructs, as well as a full-fledged verification methodology for heap-manipulating programs based on a flexible annotation

protocol, sufficient to completely verify a variety of programs and verification challenges that are representative of object-oriented idioms as used in practice (see Chapter 5). AutoProof was developed with usability and extensibility in mind: it features techniques to help debug failed verification attempts so as to offer an incremental user experience, its annotation library can be augmented with new abstract models, and its implementation can be easily adapted to accommodate changes in the input language.

3.1 Introduction

AutoProof is an auto-active verifier of functional correctness working on Eiffel programs and using Boogie [93] as a back-end verifier. Figure 5 shows the general workflow of using AutoProof. It takes compiled Eiffel programs annotated with contracts and translates them to Boogie code. The Boogie verifier is then invoked on the generated Boogie code transforming the intermediate Boogie program to verification conditions. These verification conditions are finally checked by an SMT solver (currently Z3). The result of the verification is then traced back to Eiffel code and displayed to the user.

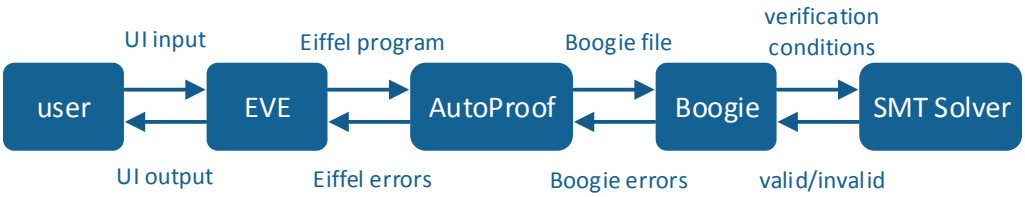


Figure 5: High-level overview of the AutoProof verification workflow.

AutoProof does modular verification on the level of routines. Each routine is checked separately and for calls inside a routine only the callee's specification is used. This allows to verify Eiffel systems piece by piece, and, when a routine body is changed, only the changed routine must be reverified. AutoProof is available online as well as integrated in EVE, the Eiffel Verification Environment IDE [62].

3.2 User Interface

AutoProof is available in two modes of operation: (1) through a graphical interface integrated in the EVE IDE and (2) as a command-line tool. In addition, there are two different web interfaces that gives access to the command-line version of AutoProof, each with different limitations.

3.2.1 *AutoProof in EVE*

EVE is the Eiffel Verification Environment, an Eiffel IDE based on EiffelStudio [57]. EVE contains various research tools including AutoProof. In EVE, AutoProof is available as a separate tool panel. Figure 6 shows EVE with the AutoProof panel open in the bottom left.

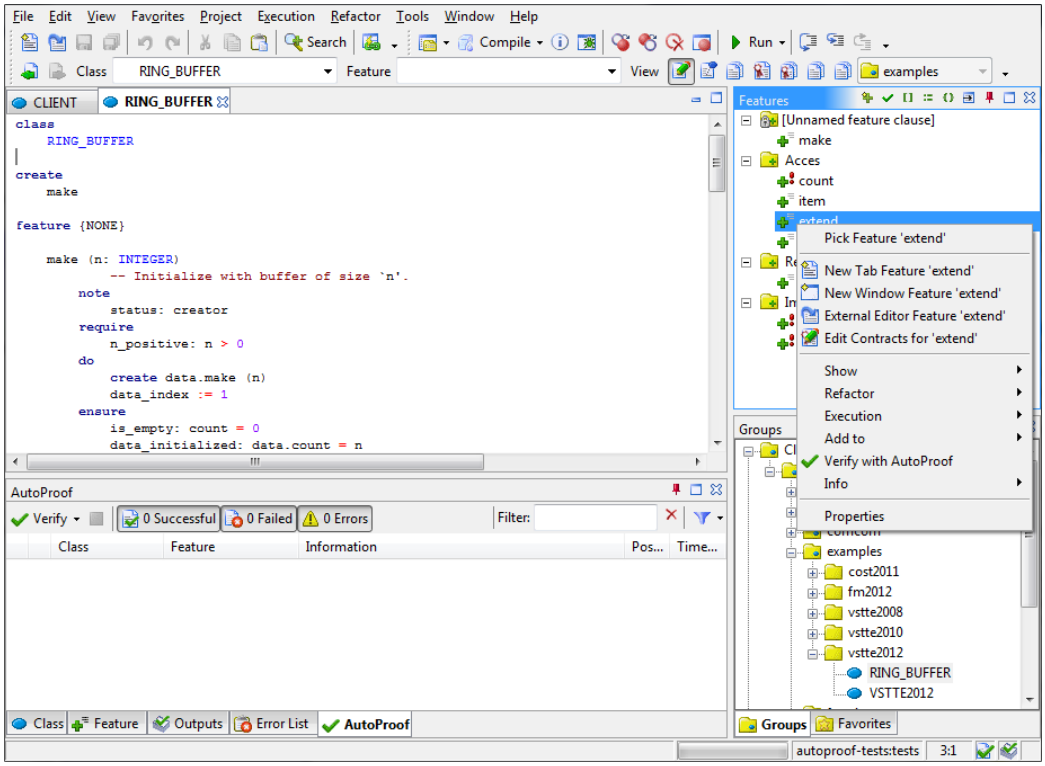


Figure 6: The AutoProof tool panel in EVE (lower left) and the context menu of a feature (right side).

Running AutoProof

The scope of the input to AutoProof can be selected by interacting with the *Verify* button:

- *Verifying a single feature*: pick-and-drop a feature on the button.
- *Verifying a single class*: either open a class in the editor and click the button or pick-and-drop a class on the button.
- *Verifying all classes in a cluster or library*: either open a cluster in the editor and click the button, pick-and-drop a cluster or library on the

AutoProof

Verify | 7 Successful | 1 Failed | 1 Error | Filter:

Class	Feature	Information	Pos...	Time...
CLIENT	reverse_inplace	Impure function 'reverse_position' cannot be used in specifications.	184	
CLIENT	invariant admissibility	Verification successful.		0.68
ANY	default_create	Verification successful.		0.09
CLIENT	swap	Verification successful.		0.13
CLIENT	wrapped_index	Verification successful.		0.09
CLIENT	is_rotation	Verification successful.		0.12
CLIENT	rotate_copy	Verification successful.		0.25
CLIENT	rotate_reverse	Variant may not decrease at this recursive call / the end of this loop body. (+1 more error)	152	0.21
		Variant may not decrease at this recursive call / the end of this loop body.	152	
		Postcondition is_rotation may be violated.	161	
CLIENT	two_way_sort	Verification successful after inlining. (see original error)		0.23
		You might need to weaken the precondition.		
		Precondition j_in_range may be violated on call to swap.	39	

Figure 7: The AutoProof panel showing the results of a verification.

button, or use the drop-down arrow on the button to verify the cluster the currently selected class is in.

- *Verifying the whole system (excluding libraries)*: use the drop-down arrow on the button and select *Verify System*

The user can also right-click any feature, class, cluster, or library and select the *Verify with AutoProof* option from the context menu (see right side of Figure 6).

AutoProof will always check that the system compiles successfully with the Eiffel compiler before starting the verification. This ensures that the code that is verified is valid Eiffel code. Only one instance of AutoProof can run at the same time, so during the execution of AutoProof the *Verify* button is disabled while the red *stop* button next to it is enabled. With the stop button the user can abort a long-running verification if necessary.

Displaying results

After the verification is finished, AutoProof displays the results of the verification in the tool panel. It displays the verification result for each individual routine separately. Figure 7 shows an example of running AutoProof. There are four possible outcomes per routine:

- Green row: The verification was successful.
- Red row: The verification failed. An error message will indicate what type of assertion was not verified correctly and, if possible, indicate the assertion tag and line number of the assertion. It is also possible that there are multiple violations for a single routine in which case the row can be expanded to show all messages.

- Yellow row: There was invalid input to AutoProof and the verification of that routine was not possible. This usually means an Eiffel construct is present that AutoProof does not support or an Eiffel assertion is invalid in another way (e.g. an impure function is used in a contract).

- Blue row: The verification was successful in the second step of two-step verification. The row can be expanded to show a possible suggestion of what the user can do to fix the original problem as well as the message of the failed first verification step.

In addition to the information about the routine and the message, the row will also indicate how long it took Boogie to verify that particular routine. The Eiffel elements such as class and feature names and the line number (if available) can be used to jump to the respective source location.

Verification options

The user can change the behavior of AutoProof by enabling or disabling the following options:

- *Two-step verification*: if enabled, a second verification attempt will be made on failed verifications.

- *Automatic inlining of routines*: if enabled, routines without postconditions will be automatically inlined.

- *Automatic loop unrolling*: if enabled, loops without a loop invariant will be automatically unrolled.

- *Generate postcondition predicates*: If enabled, AutoProof will generate postcondition predicates and corresponding axioms to verify polymorphic calls.

- *Overflow checks*: if enabled, AutoProof will check arithmetic operations for overflows.

- *Generate triggers*: if enabled, AutoProof will generate triggers for quantified arithmetic expressions in Boogie.

- *Bulk or forked verification*: the user can choose between verifying all routines in one Boogie execution or running Boogie on each routine individually.

3.2.2 AutoProof on the Command-line

Using AutoProof on the command-line is similar to using the regular Eiffel compiler. In addition to specifying the Eiffel project, the user has to specify the *-autoproof* option to launch AutoProof after the Eiffel compilation succeeded. To set options of AutoProof such as enabling or disabling arithmetic overflow checks, the user can add additional command-line flags for AutoProof (see Appendix B.4 for details). Lastly, the user can also specify which classes and routines should be verified. If no selection is given, the whole system (excluding libraries) will be verified. The following command shows an example of launching AutoProof with the project file `lcp.ecf` and overflow checking enabled on the single routine `LCP.client`:

```
ec.exe -config lcp.ecf -autoproof -overflow LCP.client
```

The output of the verification will be printed to the console analogous to the graphical version of AutoProof.

3.2.3 AutoProof on the Web

There are two web-interfaces available for AutoProof: (1) Comcom [9] and (2) E4Pubs [10]. Both of them can be used to deploy examples, provide an online editor to change the examples, and can run AutoProof on the examples using the command-line interface of AutoProof. There are different limitations and use cases though.

Comcom

Comcom is a web-interface for command-line tools. It features a list of predefined examples that the user can choose and modify as well as the possibility to write custom code from scratch and provide command-line arguments to AutoProof. Figure 8 shows the interface of Comcom after running an example verification.

The limitation of AutoProof on Comcom is the use of single-file examples. Since Eiffel only allows one class per file, the examples on Comcom have to be self-contained in one class. The user can use any class from the Eiffel base library, so it is still possible to write interesting examples and algorithms.

Due to Comcom's integration with the *rise4fun* API, AutoProof is also available on the rise4fun website [11].

cloudstudio.ethz.ch/comcom/#AutoProof

AutoProof - a Program Verifier for Eiffel

Account | Sum and Max | Longest Common Prefix | Array Sum | Doubly-Linked List | More AutoProof

```
4
5   two_way_sort (a: SIMPLE_ARRAY [BOOLEAN]): INTEGER
6       -- Sort bollean array `a' in linear time.
7       -- Returns number of False elements in array.
8     note
9       status: impure
10    require
11      a.count >= 0
12
13      modify (a)
14    local
15      i, j: INTEGER
16      old_sequence: MML_SEQUENCE [BOOLEAN]
17    do
18      from
19        i := 0
20        j := a.count
21        old_sequence := a.sequence
22      invariant
23
```

Run... -twostep

Feature	Line	Result
CLIENT.reverse_inplace	191	Validity: Impure function 'reverse_position' cannot be used in specifications.
CLIENT (invariant admissibility)		Successfully verified.
CLIENT.wrapped_index		Successfully verified.
CLIENT.is_rotation		Successfully verified.
CLIENT.rotate_reverse	152	Variant may not decrease at this recursive call / the end of this loop body.
	161	Postcondition is_rotation may be violated.
CLIENT.two_way_sort		Successfully verified after inlining. You might need to weaken the precondition. Original errors:

Figure 8: Screenshot of comcom after running an example.

E4Pubs

The second web-interface to AutoProof is *E4Pubs*, which stands for *Eiffel 4 Publications*. It offers to host a predefined set of examples that can be verified with AutoProof. Each example can—in contrast to—contain multiple classes. On the other hand, there is no possibility to write a custom example from scratch or provide command-line arguments. This web-interface is used to host examples for publications or to showcase features of AutoProof, as each example has a unique URL and the examples can also be embedded in other websites. We use E4Pubs for example to create an interactive tutorial for AutoProof [12] and showcase benchmark examples [13].

3.3 Input Language Support

AutoProof supports most of the Eiffel language as used in practice, obviously including Eiffel's native notation for contracts (specification elements) such as pre- and postconditions, class invariants, loop invariants and variants, and inlined assertions such as **`check`** (**`assert`** in other languages). Object-oriented features—classes and types, multiple inheritance, polymorphism—are fully supported, and so are imperative and procedural constructs.

Partially supported and unsupported features

A few language features that AutoProof does not currently fully support have a semantics that violates well-formedness conditions required for verification: AutoProof doesn't support specification expressions with side effects (for example, a precondition that creates an object). These are common practices in verification and recommended also in Eiffel, although programs in practice may include specification functions with side effects. It also doesn't support the semantics of **`once`** routines (similar to **static** in Java and C#), which would require global reasoning thus breaking modularity.

Other partially supported features originate in the distinction between machine and mathematical representation of types. Among primitive types, machine `INTEGER`s are fully supported (including overflows); floating-point `REAL`s are modeled as infinite-precision mathematical reals; strings are supported with a limited set of operations and without UTF support. Array and list library containers with simplified interfaces are supported out of the box. Other container types require custom specification. Agents (function objects) are partially supported, with restrictions in their specifications. The semantics of native **`external`** methods (no source code is available) is reduced to their specification. We designed a translation for exceptions based on the

latest draft of the Eiffel language standard (see Section 5.7), but AutoProof doesn't support it yet since the Eiffel compiler still only implements the former syntax for exceptions (and exceptions have very limited usage in Eiffel anyway).

The AutoProof manual (Appendix B.6) has an overview of the level of support for all Eiffel language features.

Annotations for verification

Supporting effective auto-active verification requires much more than translating the input language and specification into verification conditions. AutoProof's verification methodology relies on annotations that are not part of the Eiffel language. Users provide them by writing two kinds of constructs. Annotations that are part of assertions or other specification elements use predefined dummy features with empty implementation, which AutoProof interprets according to their semantics. Annotations of this kind include *modify* and *read* clauses (specifying objects whose state may be modified or read by a routine's body). For instance, a clause `modify (set)` in a routine's precondition denotes that executing the routine may modify objects in `set`.

Annotations that apply to whole classes or features are expressed by means of Eiffel's **note** clauses, which attach additional information that is ignored by the Eiffel compiler but is processed by AutoProof. Annotations of this kind include defining class members as *ghost* (only used in specifications), procedures as *lemmas* (outlining a proof using assertions and ghost-state manipulation), and which members of a class define its abstract *model* (to be referred to in interface specifications). For example `note status: ghost` tags as ghost the member it is attached to. A complete list of supported annotations is available in the AutoProof manual (Appendix B.7).

A distinctive trait of semantic collaboration, as available to AutoProof users, is the combination of flexible expressive annotations with useful defaults. Expressivity offers fine-grained control over the visibility of specification elements (for example, invariant clauses can be referenced individually); defaults reduce the amount of required manual annotations in many practical cases; the combination of the two is instrumental in making AutoProof usable on complex examples of realistic object-oriented programs.

Verifier's options

AutoProof *verification options* are also expressed by means of **note** clauses: users can disable generating boilerplate implicit contracts, skip verification of a specific class, disable termination checking (only verify partial correct-

ness), and define a custom mapping of a class's type to a Boogie theory file. See AutoProof's manual (Appendix B.4) for a complete list of features and options, and examples of usage.

Specification library

To support writing complex specifications, AutoProof provides a library—called MML for Mathematical Model Library—of pre-defined abstract types. These include mathematical structures such as sets, relations, sequences, bags (multisets), and maps. The MML annotation style follows the model-based paradigm [131], which helps write abstract and concise, yet expressive, specifications. MML's features are fully integrated in AutoProof by means of effective mappings to Boogie background theories. A distinctive advantage of providing mathematical types as an annotated library is that MML is *extensible*: users can easily provide additional abstractions by writing annotated Eiffel classes and by linking them to background theories using custom **note** annotations—in the very same way existing MML classes are defined. This is not possible in most other auto-active verifiers, where mathematical types for specification are built into the language syntax.

3.4 Translation

We now give a detailed description of the translation from Eiffel to Boogie using a translation function $\mathcal{T}$ and several variants of it:

- $\mathcal{T}_{\text{Name}}(x)$: translates the Eiffel class, type, or routine x to its Boogie name. The Boogie name can be used as an identifier, variable name, or procedure name. If the class or type is generic, the translated name will include an encoding of the generics as well. Two different generic deviations of the same generic base class have therefore different names in Boogie.

- $\mathcal{T}_{\text{Type}}(x)$: translates the Eiffel type x to its Boogie type.

- $\mathcal{T}_{\text{Body}}(x)$: translates the Eiffel expression x, which can have function calls with side effects, to a Boogie expression. Since Boogie expressions are side-effect free, side-effects of Eiffel expressions, such as routine calls to impure functions, are translated to Boogie **procedure** calls that are prepended to the current statement. This translation is used while translating the body of routines.

- $\mathcal{T}_{\text{Spec}}(x)$: translates the Eiffel expression x, which has to be side-effect free, to a Boogie expression. Function calls in Eiffel are translated to Boogie **function** calls. This translation type is used for translating any type of specification construct.

We use the general translation function $\mathcal{T}$ when we do not need to distinguish between special translations, e.g. for statements. An introduction to Boogie is available in Section 5.1.

3.4.1 Background Theory

AutoProof includes a background theory in all generated Boogie files. The background theory contains definitions and axioms that define the memory and framing model, as well as convenience functions and procedures for other aspects like bounded integers. We only give an overview of the full background theory here.

Memory model

To translate an object-oriented language, we need a model for object references and the heap. We define several types in Boogie to represent the necessary concepts:

```
type ref;
type Field _;
type HeapType = ⟨α⟩[ref, Field α]α;
type Type;
```

The `ref` type denotes object references and the `Field` type storage locations. We define the heap as a mapping of references and fields to the field content. The field type is generic, so we can use the content type of the field as the content type of a specific heap location. The type `Type` is used to model Eiffel types. As the static type of references cannot change during the lifetime of an object, we use a Boogie function to map references to types:

```
function type_of(r: ref): Type;
```

The Heap is modeled using a global variable `Heap` of type `HeapType`. Using a boolean ghost field `allocated` we define helper procedures for two important memory operations: allocating references and updating heap values:

```
1 procedure allocate(t: Type) returns (r: ref);
2   modifies Heap;
3   ensures ¬old(Heap[r, allocated]);
4   ensures Heap[r, allocated];
```

```
5    ensures r ≠Void;
6    ensures type_of(r) =t;
7    ensures (∀⟨α⟩ f: Field α • f ≠allocated ⟹Heap[r, f] =Default(f));

8  procedure update_heap<T>(Current: ref, field: Field T, value: T);
9    requires (Current ≠Void) ∧(Heap[Current, allocated]);
10   modifies Heap;
11   ensures Heap =old(Heap[Current, field :=value]);
```

The `allocate` procedure guarantees that the return values is a fresh reference (line 3) of the allocated type (line 6) with all fields initialized to the default value (line 7), as is the Eiffel semantics. The `update_heap` procedure guards against invalid uses of the heap by checking that the target reference is non-void and properly allocated (line 9).

Framing and invariant model

To support semantic collaboration, the background theory contains several built in ghost fields.

```
const closed: Field bool;
const owner: Field ref;
const owns: Field (Set ref);
const observers: Field (Set ref);
const subjects: Field (Set ref);
```

These fields, together with a set of axioms and helper functions, are used to define the framing model. The `closed` field denotes if an object is in a state satisfying the invariant (*closed*) or not (*open*), the `owner` field and `owns` set are used to model the ownership relationship, and the `observers` and `subjects` sets are used to model the collaboration relationship.

To model class invariants we use an uninterpreted function that will be linked to user-provided class invariants based on the type of references, and a global invariant function for the relation of closed objects and valid invariants:

```
function user_inv(h: HeapType, o: ref): bool;
function inv(h: HeapType, o: ref): bool
  { h[o, closed] ⟹user_inv(h, o) }
```

The background theory contains several Boogie procedures to manipulate the ghost fields. We show the `wrap` procedure as an example, which is used to wrap an open object `o`.

```
1  procedure wrap(o: ref);
2    requires (o ≠Void) ∧(Heap[o, allocated]);
3    requires is_open(Heap, o);
```

```
4   requires writable[o, closed];
5   requires user_inv(Heap, o);
6   requires (∀ o': ref • Heap[o, owns][o'] ⟹
7            is_wrapped(Heap, o') ∧writable[o', owner]);
8   modifies Heap;
9   ensures (∀ o': ref • old(Heap[o, owns][o']) ⟹Heap[o', owner] =o)
10  ensures is_wrapped(Heap, o)
11  ensures (∀ <T> o': ref, f: Field T •
12      ¬(o' =o ∧ f =closed) ∧¬(old(Heap[o, owns][o']) ∧ f =owner) ⟹
13           Heap[o', f] =old(Heap[o', f]));
```

The procedure requires that the object `o` is valid (non-void and allocated, line 2), open (not yet wrapped, line 3), the ghost field `closed` is writable (line 4), the specific class invariant of the object holds (line 5), and that all owned objects are wrapped with their `owner` field being writable (line 6). In turn, the procedure guarantees that all objects owned by `o` have their `owner` field set to the object being wrapped (line 9), the object will be wrapped (line 10), and that only the `closed` field of `o` and the `owner` field of the owned objects are modified (line 11).

Bounded integers

Since Boogie's integer type is unbounded, our background theory introduces helper functions to deal with machine integers of various sizes and conversion between them. For each bit size (8, 16, 32, 64) of both integers and naturals we define two helper functions and axiomatize them. As an example, we show the functions for `INTEGER_8`:

```
function is_integer_8(i: int) returns (bool) { −128 ≤i ≤127 }
function int_to_integer_8(i: int) returns (int);
axiom (∀ i: int • is_integer_8(i) ⟹int_to_integer_8(i) =i);
axiom (∀ i: int • is_integer_8(int_to_integer_8(i)));
```

These functions are used for overflow checks after each arithmetic operation and each time a numeric type conversion happens between integers or naturals of different bit sizes.

3.4.2 Types

Eiffel types are mapped to Boogie types as shown in Figure 9. The basic types of Eiffel are mapped to the corresponding basic types of Boogie. To keep the correct semantics with respect to the bounded integers in Eiffel, AutoProof inserts additional assertions and assumptions, as will be described later. Eiffel's reference types are translated to the type `ref`, which is defined

in the background theory. The MML types from the base library of EVE are all translated to specific Boogie types that are declared and axiomatized in their respective background theories.

$$
\begin{array}{lll}
\mathcal{T}_{\text{Type}}(\texttt{BOOLEAN}) & = \texttt{bool} & \\
\mathcal{T}_{\text{Type}}(\texttt{INTEGER\_X}) & = \texttt{int} & \text{where } X \in \{8, 16, 32, 64\} \\
\mathcal{T}_{\text{Type}}(\texttt{NATURAL\_X}) & = \texttt{int} & \text{where } X \in \{8, 16, 32, 64\} \\
\mathcal{T}_{\text{Type}}(\texttt{REAL\_X}) & = \texttt{real} & \text{where } X \in \{32, 64\} \\
\mathcal{T}_{\text{Type}}(\texttt{CHARACTER\_X}) & = \texttt{int} & \text{where } X \in \{8, 32\} \\
\hline
\mathcal{T}_{\text{Type}}(\texttt{X}) & = \texttt{ref} & \text{– where } X \text{ is a reference class} \\
\mathcal{T}_{\text{Type}}(\texttt{G}) & = \texttt{ref} & \text{– where } G \text{ is an unspecified generic type} \\
\hline
\mathcal{T}_{\text{Type}}(\texttt{MML\_SET [G]}) & = \texttt{Set } \mathcal{T}_{\text{Type}}(\texttt{G}) & \\
\mathcal{T}_{\text{Type}}(\texttt{MML\_SEQUENCE [G]}) & = \texttt{Seq } \mathcal{T}_{\text{Type}}(\texttt{G}) & \\
\mathcal{T}_{\text{Type}}(\texttt{MML\_MAP [K, V]}) & = \texttt{Map } \mathcal{T}_{\text{Type}}(\texttt{K})\ \mathcal{T}_{\text{Type}}(\texttt{V}) & \\
\mathcal{T}_{\text{Type}}(\texttt{MML\_BAG [G]}) & = \texttt{Bag } \mathcal{T}_{\text{Type}}(\texttt{G}) & \\
\mathcal{T}_{\text{Type}}(\texttt{MML\_INTERVAL}) & = \texttt{Set int} & \\
\mathcal{T}_{\text{Type}}(\texttt{MML\_RELATION [G, H]}) & = \texttt{Rel } \mathcal{T}_{\text{Type}}(\texttt{G})\ \mathcal{T}_{\text{Type}}(\texttt{H}) &
\end{array}
$$

Figure 9: Translation of Eiffel types to Boogie types.

3.4.3 Classes and Features

Classes

The declaration of an Eiffel class—**class** C **inherit** B—is translated to the Boogie code shown in Figure 10. For each class a constant value of type Type is declared. This constant is used to represent the type value of the class. An axiom is generated that models the inheritance based on Boogie's partial order operator. To express the class invariant C_{inv} a function $\mathcal{T}_{\text{Name}}(C)$.user_inv() is generated, having the translation of the combined (**and**ed) class invariant as its body. Two axioms are generated that relate the specific class invariant to the global user_inv function based on the type of a reference.

Attributes

A constant in Boogie is generated for each attribute in Eiffel. This constant is used to access the heap location of the attribute. Since the name of the attribute is based on the static type, additional axioms are generated relating

```
1 const unique T_Name(C): Type;
2 axiom (T_Name(C) <: T_Name(B));
3 function T_Name(C).user_inv() { T_Spec(C_inv) }
4 axiom (∀ h: HeapType, c: ref •attached_exact(h, c, T_Name(C)) ⟹
5              (user_inv(h, c)=T_Name(C).user_inv(h, c)));
6 axiom (∀ h: HeapType, c: ref •attached(h, c, T_Name(C)) ⟹
7              (user_inv(h, c)⟹T_Name(C).user_inv(h, c)));
```

Figure 10: Translation of Eiffel **class** declarations.

inherited attributes to their ancestor versions. Given an attribute declaration `a: T` in class `C` and a subclass `D` of `C`, the following Boogie code will be generated for the two attributes `C.a` and `D.a`:

```
const T_Name(C.a): Field T_Type(T);
const T_Name(D.a): Field T_Type(T);
axiom T_Name(C.a) = T_Name(D.a);
```

An additional axiom will be generated depending on the type of the attribute. Given an attribute `a: T` in class `C` an axiom enforcing a type specific function is generated:

```
axiom (∀ h: HeapType, o: ref •attached(h, o, T_Name(C)) ⟹T_TF(T));
```

The type function $\mathcal{T}_{\text{TF}}(T)$ is based on the type `T` of the attribute according to Figure 11. The *exact* variants for reference types are used whenever AutoProof can determine that the dynamic type and static type are identical (e.g. if `T` is *frozen*). For integer types an axiom is generated by AutoProof to restrict the range of the value.

Type `T`	$\mathcal{T}_{\text{TF}}(T)$
attached `T`	attached(h, h[o,$\mathcal{T}_{\text{Name}}(C.a)$], $\mathcal{T}_{\text{Name}}(T)$)
exact **attached** `T`	attached_exact(h, h[o,$\mathcal{T}_{\text{Name}}(C.a)$], $\mathcal{T}_{\text{Name}}(T)$)
detachable `T`	detachable(h, h[o,$\mathcal{T}_{\text{Name}}(C.a)$], $\mathcal{T}_{\text{Name}}(T)$)
exact **detachable** `T`	detachable_exact(h, h[o,$\mathcal{T}_{\text{Name}}(C.a)$], $\mathcal{T}_{\text{Name}}(T)$)
`INTEGER_X`	is_integer_X(h[o, $\mathcal{T}_{\text{Name}}(C.a)$])
`NATURAL_X`	is_natural_X(h[o, $\mathcal{T}_{\text{Name}}(C.a)$])
`CHARACTER_X`	is_integer_X(h[o, $\mathcal{T}_{\text{Name}}(C.a)$])

Figure 11: Type property functions.

Routines

Routines are translated in two parts: the signature is translated to a Boogie **procedure** and the routine body to a Boogie **implementation**. The translation of a general Eiffel routine r in a class C is shown in Figure 12.

$$\mathcal{T}\left(\begin{array}{l} \texttt{r}\ (a_1\colon T_1;\ \cdots;\ a_n\colon T_n)\colon T_{res} \\ \quad \textbf{require}\ r_{pre}\ \textbf{do}\ B \\ \quad \textbf{ensure}\ r_{post}\ \textbf{end} \end{array}\right) =$$

```
1  procedure T_Name(C.r)(Current: ref, a_1: T_Type(T_1), ···, a_n: T_Type(T_n))
2        returns (Result: T_Type(T_res) where T_TF(T_res));
3    free requires attached_exact(Heap, Current, T_Name(C));
4    free requires T_TF(T_1) ··· T_TF(T_n);
5    modifies Heap;
6    requires T_Spec(r_pre)
7    ensures T_Spec(r_post)
8    requires Frame#Subset(
9          modify.T_Name(C.r)(Heap, Current, a_1, ···, a_n), writable);
10   free ensures same_outside(old(Heap), Heap,
11         modify.T_Name(C.r)(old(Heap), Current, a_1, ···, a_n));

12 implementation T_Name(C.r)(Current: ref, a_1: T_Type(T_1), ···,
13         a_n: T_Type(T_n)) returns (Result: T_Type(T_res))
14 { T(B) }
```

Figure 12: Translation of Eiffel routines.

We model the object-oriented Eiffel in procedural Boogie by adding the **Current** object as the first argument to the procedure. For the target object and all routine arguments we model the Eiffel semantics by adding free preconditions using the type property functions (lines 3 and 4). The type property of the result is also added using the Boogie **where** clause (line 2). Each of the routine's pre- and postconditions are translated directly to a **requires** or **ensures** clause (lines 6 and 7). For each routine a Boogie function is generated to represent the frame of the routine. The function expresses which heap locations the routine is allowed to change. We use this function in the signature of the routine to express the frame condition in two parts: (1) the locations the routine is allowed to change needs to be writable (line 8) and (2) all locations outside of the routine's frame remain unchanged after the routine exits (line 10). An implementation block is generated for the body of the routine with the same signature as the corresponding procedure and the translation of the routine's body as the actual implementation

(line 14).

Figure 12 shows the general translation of routines. There are several special cases for specific types of routines. *Creation procedures* have an additional precondition that all attributes of the class are set to their default value. Finally, a Boogie **function** is created for *side-effect free functions* that can be used in specifications:

$$
\begin{array}{l}
\textbf{function}\ \texttt{fun}.\mathcal{T}_{\text{Name}}(C.r)(\texttt{heap}\colon\ \texttt{HeapType};\ \texttt{current}\colon\ \texttt{ref}, \\
\quad a_1\colon\ \mathcal{T}_{\text{Type}}(T_1), \cdots, a_n\colon\ \mathcal{T}_{\text{Type}}(T_n))\ \textbf{returns}\ (\mathcal{T}_{\text{Type}}(T_{res}));\\
\textbf{axiom}\ (\forall\ \texttt{h}\colon\ \texttt{HeapType},\ \texttt{c}\colon\ \texttt{ref},\ a_1\colon\ \mathcal{T}_{\text{Type}}(T_1), \cdots, a_n \bullet \\
\quad \mathcal{T}_{\text{Spec}}(r_{pre}) \Longrightarrow (\texttt{fun}.\mathcal{T}_{\text{Name}}(C.r)(\texttt{h},\ \texttt{c},\ a_1, \cdots, a_n) = \mathcal{T}_{\text{Spec}}(r_{post})));
\end{array}
$$

3.4.4 Instructions

Sequential composition

Sequential statements in Eiffel are translated to sequential statements in Boogie: $\mathcal{T}(I_1; I_2) = \mathcal{T}(I_1); \mathcal{T}(I_2)$.

Conditional

Although Boogie offers a cascading **if** statement[1] we have to use the nested conditionals for the translation from Eiffel because the condition in a Boogie **if** statement cannot have a side-effect. The **elseif** condition in Eiffel can be a function call with side-effects though, so we have to be able to call a Boogie procedure before evaluation each cascading condition. The translation for conditionals is therefore as follows:

$$
\mathcal{T}\left(\begin{array}{l}
\textbf{if}\ c_1\ \textbf{then}\ B_1 \\
\textbf{elseif}\ c_2\ \textbf{then}\ B_2 \\
\textbf{else}\ B_3 \\
\textbf{end}
\end{array}\right) = \begin{array}{l}
\textbf{if}\ (\mathcal{T}_{\text{Body}}(c_1))\ \{\ \mathcal{T}(B_1)\ \} \\
\textbf{else}\ \{ \\
\quad \textbf{if}\ (\mathcal{T}_{\text{Body}}(c_2))\ \{\ \mathcal{T}(B_2)\ \} \\
\quad \textbf{else}\ \{\ \mathcal{T}(B_3)\ \} \\
\}
\end{array}
$$

Loop

The translation of loops is among the more complex. Loops can have extensive specifications similar to routines: loop invariants, loop variant, and loop modifies clauses. The translation of a loop with exit condition c, loop body L, loop invariants $i_1, ...i_n$, and loop variant v is given in Figure 13.

The loop structure is translated into three labeled blocks `head`, `body`, and `end`. The `head` contains the loop invariants (Lines 3 to 5), which will be

[1] In Boogie you can write **if** $(c_1)\{\ \cdots\}$ **else if** $(c_2)\{\ \cdots\}$ **else** $\{\ \cdots\}$.

```
                                                  7  body:
                                                  8    assume ¬(T_Spec(c));
                                                  9    v_old := T_Spec(v);
from A                                            10   T(L);
invariant                                         11   assert 0 ≤ v_old;
  i_1                                             12   assert T_Spec(v) < v_old;
  ⋮                                               13   goto head;
  i_n                                             14 end:
until c                                           15   assume T_Spec(c);
loop L
variant v
end
```

$$\mathcal{T}(\quad) = \quad$$

```
1   T(A);
2 head:
3   assert T_Spec(i_1);
4   ⋮
5   assert T_Spec(i_n);
6   goto body, end;
```

Figure 13: Translation of Eiffel **from**-loops.

checked on entry and after every iteration. The loop invariants use the side-effect free expression translation $\mathcal{T}_{\text{Spec}}(x)$, so only pure functions are allowed in loop invariants (as with other specifications). The last statement of the head block is a non-deterministic jump instruction, jumping both to the `body` and `end` blocks. The first statement in the `body` block is an **assume** statement with the negated exit condition (Line 8). Since Eiffel loops use an **until** exit condition, the negation signifies loop continuation. The second statement in the loop evaluates the loop variant before the loop body is executed and stores it in a local variable (Line 9). Next follows the translation of the loop body and checking of the loop variant. The first assertion checks that the loop variant is bounded (Line 11) and the second assertion checks that the loop variant after execution of the body is strictly smaller then before (Line 12). The last statement of the loop body jumps back to the loop head where the loop invariant is checked again. The `end` block assumes the exit condition.

Inspect

The Eiffel **inspect** statement is a *switch*-statement on integer values. Each individual case is defined by a statically defined continuous interval. The compiler enforces that the individual intervals are not overlapping.

The translation to Boogie, shown in Figure 14, works by having a labeled block for each individual case and the **else** branch. After evaluating the inspect value the goto on line 2 jumps to all labeled blocks nondeterministically. Each block assumes the condition that the inspect value is in this particular interval (e.g. line 6), followed by the translation of the block body and a jump to the `end` label. The block of the else-branch assumes the value is outside all other intervals (line 15).

$\mathcal{T}($

```
inspect v
when l1..u1
  then B1
⋮
when ln..un
  then Bn
else B0
end
```

$) =$

```
1  s := T_Body(v)
2  goto case_else,
3    case_1, ···,
4    case_n;
5 case_1:
6  assume l1 ≤s ≤u1;
7  T(B1)
8  goto end;
9  ⋮
10 case_n:
11  assume ln ≤s ≤un;
12  T(Bn)
13  goto end;
14 case_else:
15  assume ¬(l1 ≤s ≤u1) ∧
16    ··· ∧ ¬(ln ≤s ≤un);
17  T(B0)
18  goto end;
19 end:
```

Figure 14: Translation of Eiffel **inspect**-statements.

Assignment

The translation of assignments depend on the type of the target. An assignment target in Eiffel can only be an attribute of the current object, a local variable, or the special local variable **Result** in functions. The **Result** variable behaves in this context like a local variable, so we can distinguish two cases in total. Assuming the assignment takes place in a routine of class C, the translations are as follows:

$$\mathcal{T}(\texttt{o} := \texttt{v}) = \begin{cases} \mathcal{T}_{\text{Name}}(o) := \mathcal{T}_{\text{Body}}(v); & (L) \\ \texttt{update\_heap}(\texttt{Current}, \mathcal{T}_{\text{Name}}(C.o), \mathcal{T}_{\text{Body}}(v)); & (A) \end{cases}$$

We can use the direct assignment to local variables in Boogie for local variables (L). For attributes — case A —, we use the procedure `update_heap` from the background theory. The preconditions of `update_heap` serves as a validity checks for updates and the postcondition describes the effect of the assignment on the heap.

Call

Call statements in Eiffel can only be procedure calls, as it is not allowed to call a function and ignore the return value. Given a call to procedure p on object o of type T, the translation to Boogie is:

$$\mathcal{T}(\texttt{o.p}\ (a_1, \cdots, a_n)) = \textbf{call}\ \mathcal{T}_{\text{Name}}(T.p)(\mathcal{T}_{\text{Body}}(o), \mathcal{T}_{\text{Body}}(a_1), \cdots, \mathcal{T}_{\text{Body}}(a_n));$$

The translation is straightforward: using the type T and procedure name p we generate the name of the Boogie procedure and translate all arguments a_i including the target object o.

Creation

Object creation can be broken down to allocating the object, calling the creation procedure, and attaching the newly created object to its target. Creating an object o of type T using creation procedure p is therefore translated as follows:

$$\mathcal{T}(\texttt{\textbf{create} o.p}\ (a_1, \cdots, a_n)) =$$
$$\texttt{\textbf{call} l} := \texttt{allocate}(\mathcal{T}_{\text{Name}}(T));$$
$$\texttt{\textbf{call} create.}\mathcal{T}_{\text{Name}}(T.p)(\texttt{l}, \mathcal{T}_{\text{Body}}(a_1), \cdots, \mathcal{T}_{\text{Body}}(a_n));$$
$$\mathcal{T}(o := l)$$

First, a new reference of the correct type is allocated using the `allocate` procedure introduced in the background theory. This fresh reference is assigned to a local variable l. Then, the creation procedure is called using the naming convention of prepending *create* to the translated name of the creation procedure. At the end, the local variable is assigned to the target entity using the translation previously discussed for assignment statements.

Check

Check instructions are translated to Boogie's **assert** statement. For each individual check expression an assert statement is generated. There is one special case: when the *tag* of the check expression is *assume*, AutoProof will generate a Boogie **assume** statement instead. This allows the user to insert regular Eiffel expressions as assumptions in the Boogie code. Recently, Eiffel has introduced the *guard* instruction, which is a check instruction that has a body. The idea of the guard instruction is a check instruction that will always be executed, no matter what the contract execution settings are. Since in Boogie there is no such thing as a disabled contract-check, we translate the guard instruction in the same way as the check instruction. The detailed translations are as follows:

$$\mathcal{T}(\texttt{\textbf{check} tag: e \textbf{then} B \textbf{end}}) = \mathcal{T}(\texttt{\textbf{check} tag: e \textbf{end}});\ \mathcal{T}(B)$$
$$\mathcal{T}(\texttt{\textbf{check} tag: e \textbf{end}}) = \begin{cases} \texttt{\textbf{assert}}\ \mathcal{T}_{\text{Body}}(e); & \text{if tag} \neq \text{assume} \\ \texttt{\textbf{assume}}\ \mathcal{T}_{\text{Body}}(e); & \text{if tag} = \text{assume} \end{cases}$$

3.4.5 Expressions

Several aspects of translating expressions depend on the expression appearing in a specification or an executable statement.

- *Side effects*: when an expression contains side effects (object creation, routine calls) we translate the side effect as additional Boogie instruction that will be prepended to the location of the currently translated expression. This only works for expressions in the routine body though, if an expression with a side-effect appears in a specification we report this as invalid input.

- *Safety checks*: safety checks enforce the language semantics, for example overflow checks or checking that a target is attached. When an expression in a body triggers a safety check, the check is prepended to the current location as an assert instruction. During the translation of specifications such as pre- or postconditions, safety checks are added as implicit specifications of the same specification type.

- When expressions are translated with either the $\mathcal{T}_{\text{Body}}$ or $\mathcal{T}_{\text{Spec}}$ functions, its subexpressions typically use the same translator function. We only distinguish between the two in the description of expression translations when necessary and otherwise use the general translation function $\mathcal{T}$.

Entity mapping

In the translation of expressions we use a mapping function for special entities. For example the Eiffel **`Current`** entity inside a routine body needs to be translated to `Current` (the name of the argument of the Boogie procedure), whereas inside the axiom for the class invariant the translation needs to be `c` (the bound variable of the axiom's quantifier representing the **`Current`** object). A mapping is not only necessary for Eiffel entities but also for the global `Heap` variable and the prestate variable **`old`**(`Heap`) in Boogie, which, depending on the translation context, might also be translated to a bound variable of a quantified expression. In the remainder of this section we use the mapping function $\mathcal{M}$ to translate an entity to the context-dependent name in Boogie.

Access

All local entities need to use the mapping function $\mathcal{M}$ for the translation to Boogie.

$$
\begin{aligned}
\mathcal{T}(\texttt{\textbf{Current}}) &= \mathcal{M}(\texttt{\textbf{Current}}) \\
\mathcal{T}(\texttt{\textbf{Result}}) &= \mathcal{M}(\texttt{\textbf{Result}}) \\
\mathcal{T}(\texttt{l}) &= \mathcal{M}(l), \text{ where } l \text{ is local variable} \\
\mathcal{T}(\texttt{a}) &= \mathcal{M}(a), \text{ where } a \text{ is an argument}
\end{aligned}
$$

The access to an attribute is translated as an access to the heap. An unqualified access to an attribute `a` is translated as if the access is qualified using **`Current`**`.a`. The translation of an attribute access `o.a`, where `a` is an attribute of an object `o` of type `T`, is the following:

$$\mathcal{T}(\texttt{o.a}) = \mathcal{M}(\texttt{Heap})[\mathcal{T}(o), \mathcal{T}_{\text{Name}}(T.a)]$$

The `Heap` name depends on the context of the translation and therefore uses the mapping function $\mathcal{M}$.

Constants

Constants of basic types in Eiffel—booleans, integers, and floating points—are translated directly to their Boogie counterpart. Characters are translated using their integer character code and the Eiffel **`Void`** value is translated to its counterpart defined in the background theory.

$$\begin{array}{ll}
\mathcal{T}(\texttt{True}) & = \texttt{true} \\
\mathcal{T}(\texttt{False}) & = \texttt{false} \\
\mathcal{T}(\texttt{x}) & = \texttt{x}, \text{ where } \texttt{x} \text{ is an integer} \\
\mathcal{T}(\texttt{'x'}) & = \texttt{y}, \text{ where } \texttt{y} \text{ is the character code of } \texttt{'x'} \\
\mathcal{T}(\texttt{x.y}) & = \texttt{x.y}, \text{ where } \texttt{x.y} \text{ is a floating point value} \\
\mathcal{T}(\texttt{Void}) & = \texttt{Void}
\end{array}$$

Operations

For operations of basic types we reuse the corresponding Boogie operations.

- The Eiffel boolean operations **`not`**, **`and`**, **`or`**, **`xor`**, and **`implies`** are translated to $\neg$, $\wedge$, $||$, $\neq$, and $\Longrightarrow$, respectively.
- The Eiffel integer operations $+$, $-$, $*$, $//$ (quotient of integer division), and $\backslash\backslash$ (remainder of integer division) are translated to $+$, $-$, $*$, `div`, and `mod`, respectively. Safety checks are added to ensure the result is in the bounds of the machine representation when overflow checks are enabled.
- The Eiffel floating point operations $+$, $-$, $*$, and $/$ are translated to their direct equivalents.
- The Eiffel comparison operations on integers and floating point values $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$ are translated to their direct equivalents.
- The Eiffel comparison operations on object references $=$ and $\neq$ are translated to their direct equivalents.

One can also define custom prefix or infix operators in Eiffel, internally represented as routine calls. Their translation uses a call to the routine implementing the operation.

Function calls

The translation of a function call `o.f` $(a_1, \cdots, a_n)$ where `o` is of type `T` is different in the body of a routine or the specification. If the call appears in the body of a routine, it is replaced with a fresh local variable; a side effect is created that calls the function and assigns the result to the fresh variable. The translation of the call is therefore:

$$\mathcal{T}_{\text{Body}}(\texttt{o.f}\ (a_1, \cdots, a_n)) = \texttt{l}, \text{ where } \texttt{l} \text{ is a fresh local variable}$$

And the created side effect is:

$$\textbf{call}\ \texttt{l} := \mathcal{T}_{\text{Name}}(T.f)(\mathcal{T}_{\text{Body}}(o), \mathcal{T}_{\text{Body}}(a_1), \cdots, \mathcal{T}_{\text{Body}}(a_n));$$

If the call appears in a specification construct, the function needs to be side-effect free (otherwise an invalid input violation will be triggered), and there exists a functional representation of the routine. We use the functional representation in the translation of the call:

$$\mathcal{T}_{\text{Spec}}(\texttt{o.f}\ (a_1, \cdots, a_n)) = \texttt{fun}.\mathcal{T}_{\text{Name}}(T.f)(\mathcal{T}_{\text{Spec}}(o), \mathcal{T}_{\text{Spec}}(a_1), \cdots, \mathcal{T}_{\text{Spec}}(a_n));$$

Nested expression

For all nested expressions `a.b`, where `a` is a reference type, a safety check **assert** $\mathcal{T}(\texttt{a}) \neq$`Void` is added to ensure that the reference `a` is attached to an object.

Creation expression

The creation of an object is a side-effect, therefore, this type of expression is only allowed in the body of a routine. We introduce a fresh local variable `l` that is used to translate the creation expression like a creation instruction on `l`:

$$\mathcal{T}(\textbf{create}\ \{\texttt{T}\}.\texttt{make}\ (a_1, \cdots, a_n)) = \begin{cases} \mathcal{T}_{\text{Body}}(\texttt{l}) & \text{in body} \\ \text{error} & \text{in specification} \end{cases}$$

In addition the following side effect is created:

$$\mathcal{T}_{\text{Body}}(\textbf{create}\ \texttt{l}.\texttt{make}\ (a_1, \cdots, a_n)).$$

Conditional expression

Eiffel's conditional expression can be directly translated to Boogie's conditional expression:

$$\mathcal{T}(\texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ ) = \texttt{if}\ (\mathcal{T}(c_1))\{\mathcal{T}(e_1)\}\ \texttt{else}\ \{\mathcal{T}(e_2)\}$$

Old expression

To translate **old** expressions we take advantage of the entity mapping for the Boogie `Heap` variable. The regular heap mapping is replaced with the prestate value **old**(`Heap`) for the translation of the subexpression.

$$\mathcal{T}(\texttt{old}\ \texttt{e}) = \mathcal{T}(e)\langle\mathcal{M}(\texttt{Heap}) := \mathcal{M}(\texttt{old}(\texttt{Heap}))\rangle$$

Across expression

Eiffel's loop expressions (**across**..**all** / **across**..**some**) are translated to universal and existential quantifiers in Boogie. The translation follows a general scheme, but the detailed translation depends on the container type of the iteration. We show the translation of a universal quantification over integer intervals and existential quantification over container types containing objects as an example.

$$\begin{array}{ll}\texttt{across}\ \texttt{l}|..|\texttt{u}\ \texttt{as}\ \texttt{i}\ \texttt{all} f(i)\texttt{end} & = \forall\ \texttt{k}\colon \texttt{int} \bullet\ \texttt{l} \leq \texttt{k} \leq \texttt{u} \Longrightarrow \mathcal{T}(f(k)) \\ \texttt{across}\ \texttt{c}\ \texttt{as}\ \texttt{o}\ \texttt{some} f(o)\texttt{end} & = \exists\ \texttt{k}\colon \texttt{ref} \bullet\ \texttt{k} \in \texttt{set}(\texttt{c}) \wedge \mathcal{T}(f(k))\end{array}$$

The across expression over integer intervals uses the interval boundaries to restrict a bound integer variable in a Boogie quantifier. Across expressions over container types use a set representation of the container contents to restrict a bound reference variable. Access to the iteration cursor is translated by accessing the bound variable in both cases.

Object test

To check if an entity conforms to a specific type at runtime, Eiffel offers the object test. In the translation to Boogie, we use the `type_of` function defined in the background theory and the partial order operator $<:$ to express this check.

$$\mathcal{T}(\texttt{attached}\ \{T\}\ e) = \texttt{o} \neq \texttt{Void} \wedge \texttt{type\_of}(\mathcal{T}(o)) <:\ \mathcal{T}_{\text{Name}}(T)$$

3.5 Implementation

AutoProof is implemented as a library in EVE. The AutoProof library consists of around 160 classes with a total of more than 25'000 lines of code. The AutoProof library only relies on the Eiffel compiler and handles the translation to and from Boogie. In addition there is the user interface code that deals with the user interaction via GUI and command-line, i.e. starting AutoProof and displaying the results to the user.

3.5.1 Main interface

The main interface to AutoProof is a single class: `E2B_AUTOPROOF`. The prefix `E2B` stands for *Eiffel to Boogie*[2]. Figure 15 shows the minimal code necessary to launch AutoProof. After creating an instance of the AutoProof class, clients use the routines `add_class` and `add_feature` to add classes and routines that should be verified. After adding a notification agent via `add_notification`, a call to `verify` will start the process of translating the added classes and features to Boogie and starts the verification. Once the verification is finished, the notification agent will be called with a result object of type `E2B_RESULT`. The result object contains a list of verification results of both successful and failed verifications, and possibly a list of semantic and execution errors.

3.5.2 Verification task

Internally AutoProof uses the EiffelStudio ROTA task system, which allows to run tasks asynchronously without the use of multithreading. This is achieved by breaking the implementation of tasks down to small steps that are called during the idle time of the application. The ROTA system is executed in the user interface loop, making it important to keep the processing time of each individual step to a minimum as otherwise the user interface will block during the execution of AutoProof. Using the ROTA task system instead of real threads is necessary though as the core compiler data structures that AutoProof used during the translation to Boogie are not thread safe. Using the ROTA system guarantees that no race conditions occur.

Figure 16 shows the AutoProof tasks and their relationship. There are two different high-level verification tasks in AutoProof and both are split into the same subtasks. The two main tasks—*bulk verification* and *forked verification*—differ in the way they handle the granularity of the Boogie ver-

[2]Eiffel does not support namespaces, the prefix scheme is used to avoid name clashes.

```
verify_class (a_class: CLASS_C)
    -- Verify class 'a_class' with AutoProof.
  local
    autoproof: E2B_AUTOPROOF
  do
    create autoproof.make
    autoproof.add_class (a_class)
    autoproof.add_notification (agent notify)
    autoproof.verify
  end

notify (a_result: E2B_RESULT)
    -- Handle result from AutoProof.
  do
    -- evaluate result
  end
```

Figure 15: Minimal code to run AutoProof

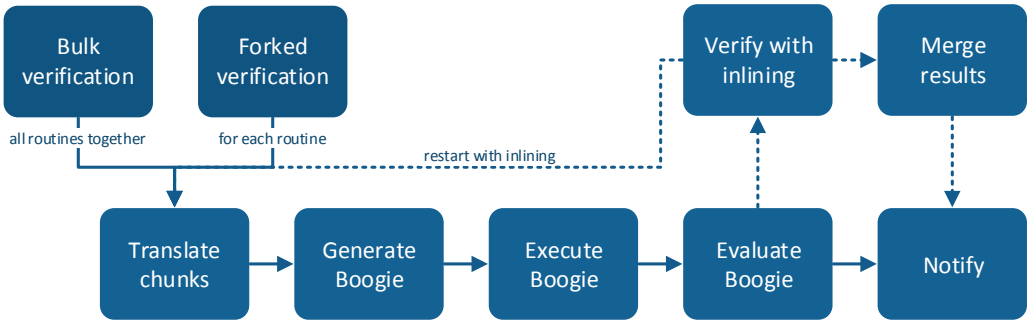


Figure 16: Workflow of verification main tasks and sub-tasks.

ification. In bulk mode all input is translated into a single Boogie file; results are fed back to users when verification of the whole input has completed. Using AutoProof in bulk mode minimizes translation and Boogie invocation overhead but provides feedback synchronously, only when the whole batch has been processed. In contrast, forked mode offers asynchronous feedback: each input routine (and implicit proof obligations such as class invariant admissibility checking) is translated into its own self-contained Boogie file; parallel instances of Boogie run on each file and results are fed back to users asynchronously as soon as any Boogie process terminates.

Both types of main tasks use the same sub-tasks to do the actual verifi-

cation:

1. *Translate chunk*: Translates part of the Eiffel input to an intermediate AST representation of Boogie. This task is executed as many times as necessary to translate the whole input.

2. *Generate Boogie*: Takes the intermediate AST and generates Boogie code from it. This Boogie code is combined with the background theory to a single Boogie file.

3. *Execute Boogie*: Executes Boogie on the generated file and waits for its termination. The Boogie process is spawned in a separate process.

4. *Evaluate Boogie*: Evaluates the output from Boogie, traces the findings back to Eiffel, and builds the result object.

5. *Verify with inlining*: If two-step verification is enabled and there where verification errors, this task restarts the verification of the failed routines with inlining and unrolling enabled by going back to the *translate chunk* task.

6. *Merge results*: This task combines the results of two runs of two-step verification.

7. *Notify*: Finally, this task notifies the client of AutoProof by calling the notification agents with the verification result.

3.5.3 Intermediate AST

For the translation of Eiffel to Boogie, AutoProof uses an intermediate abstract syntax tree (AST). The intermediate AST models an intermediate verification language like Boogie. The AST contains top-level *declarations* of variables, constants, functions, axioms, procedures and procedure bodies. Procedure bodies contain *statements* like assertions, assignments, procedure calls, and so on. Many statements and top-level declarations are composed of *expressions*: values, operations, function calls, quantifiers, and more. This part of the AutoProof library is self-contained and could be used by other tools to generate Boogie code.

Using an intermediate AST serves multiple goals:

1. Making the implementation of the translation from Eiffel simpler.

2. Making it easier to react to changes in the Boogie verifier.

3. Making it possible to implement operations on the intermediate AST.

The first goal has been clearly met. By building an AST, the code of the translator classes are more readable and maintainable, as the programmer is free to think about the generated program in abstract terms, and does not have to take specific syntax or matching braces into account. The alternative—generating Boogie code directly—quickly becomes unmanageable (a previous version of AutoProof did generate Boogie code directly).

The second goal has been successfully met as well. Since the generation of Boogie code from the intermediate AST is concentrated in a single class it is not difficult to react to changes or extensions in the Boogie syntax.

The last goal is possible with the current design, although it is not being exploited actively at the moment.

3.5.4 Translating Eiffel

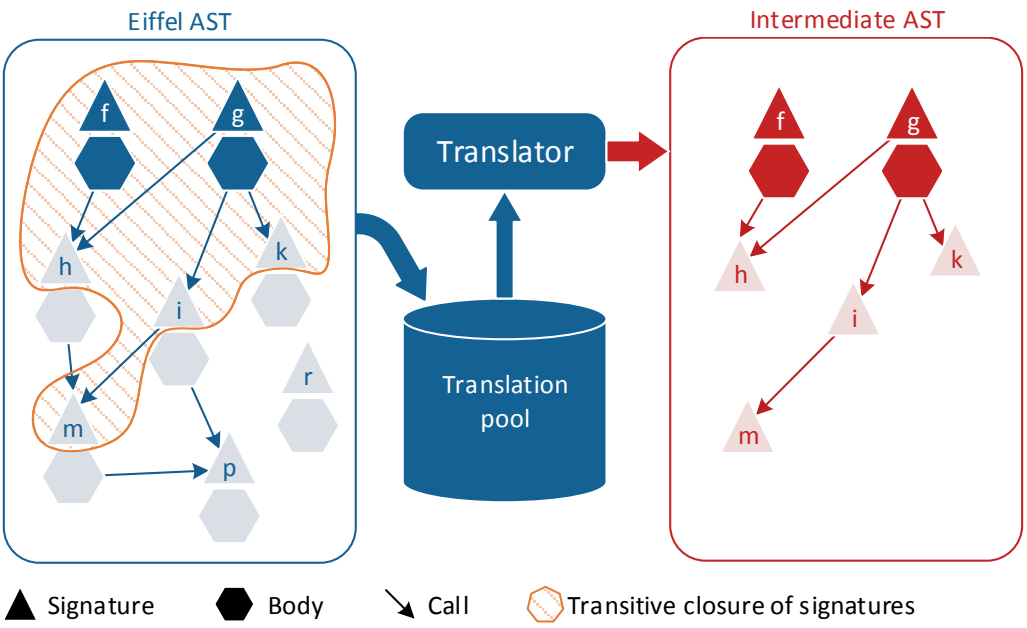


Figure 17: Translation workflow using a translation pool.

The key task of AutoProof is translating Eiffel to the intermediate AST representation. For this, AutoProof uses the notion of *translation unit* and *translation pool*. A translation unit represents an item that needs to be translated from Eiffel to Boogie and the translation pool keeps track of all translation units that need to be translated or are already translated.

The general translation workflow is shown in Figure 17. The process is started by adding the user input (dark blue shapes) from the Eiffel universe to the translation pool; for all routines that are going to be verified, the intermediate AST representation of the signature and implementation of the routine will be generated (dark red shapes). To do this, the translator (1) takes out one untranslated unit from the pool, (2) generates the intermediate AST representation of that unit, (3) marks the translation unit as translated, and finally (4) if there are untranslated units left in the pool goes back to (1). When the translator encounters an Eiffel routine call, an attribute access, or use of an Eiffel type (e.g. as the type of a local variable), that information of the Eiffel universe needs to be translated as well. These elements are added to the translation pool whenever needed. For routine calls only the signature of the routine will be translated, as AutoProof does modular verification. Through this process the translation pool grows until the transitive closure of the signature of referenced features is reached.

The different translation units in AutoProof are:

- *Type*: generates type constant and inheritance axioms for an Eiffel type.
- *Class*: generates the invariant admissibility check for an Eiffel class.
- *Invariant*: generates the invariant predicate and corresponding axioms for an Eiffel type.
- *Attribute*: generates field constant for an Eiffel attribute and additional axioms depending on the type of the attribute.
- *Routine signature*: generates the signature of a regular Eiffel routine.
- *Routine implementation*: generates the implementation of a regular Eiffel routine.
- *Creator signature*: generates the signature of an Eiffel creation routine.
- *Creator implementation*: generates the implementation of an Eiffel creation routine.
- *Functional routine*: generates a Boogie function and corresponding axioms for a pure Eiffel function.
- *Precondition predicate*: generates a predicate and corresponding axioms for the precondition of an Eiffel routine.
- *Postcondition predicate*: generates a predicate and corresponding axioms for the postcondition of an Eiffel routine.

- *Logical signature*: generates the signature of a logical routine.
- *Contract check*: generates a check for the validity of the read frame or write frame of an Eiffel routine.
- *Read frame*: generates a Boogie function for the read frame of an Eiffel routine.
- *Write frame*: generates a Boogie function for the write frame of an Eiffel routine.
- *Variants*: generates a Boogie function for the decreases function of a recursive Eiffel routine.

3.5.5 Interaction with Boogie

Once the Eiffel code is translated to the intermediate AST, we can generate the Boogie file and launch the Boogie verifier. We use the visitor pattern [74] to generate the Boogie code; the visitor traverses the AST and generates the appropriate Boogie code for each AST node. The generated Boogie code is prepended with the background theory file that contains the basic definitions and axioms and is written to a file.

Launching the Boogie verifier is done through `E2B_EXECUTABLE`, an abstract class that can be implemented through different execution schemes. The standard way of using Boogie is by launching it as a separate process. In addition, we have implemented a remote execution, where the Boogie file is sent over the network to a server which launches Boogie, and then sends back the result. Boogie files are generally quite small and do not take long to send over the network, verification on the other hand can be resource-intensive, this could make it beneficial to run the verification for multiple users on a server instead of the user machine.

When Boogie finishes, AutoProof parses the output and generates an `E2B_RESULT` object and returns it to the client of AutoProof.

3.5.6 Tracing result to Eiffel

In order to trace Boogie errors to the originating Eiffel source code, AutoProof uses structured names for Boogie procedures and inserts structured comments into the Boogie code. For an Eiffel routine `r` in class `C`, the Boogie procedure generated for verifying this Eiffel routine is called `C.r`[3] and for

[3]Boogie allows symbols such as the dot or hash in identifiers

each assertion that is generated in the Boogie code a structured comment similar to the following comment is appended:

```
// type: termination tag: variant_decreases line: 138
```

When Boogie reports its results, the Boogie procedure name can be parsed and mapped to the original Eiffel routine. For each assertion violation the Boogie source code is used to read the type of assertion and other available information such as line number. In addition to *type*, *tag*, and *line*, the assertion comments can also contain information about called routines (for precondition violations) or if an assertion was automatically generated (e.g for semantic collaboration defaults). AutoProof displays this information through the user interface and in the error message.

AutoProof has a feature of associating a result handler (in form of an Eiffel agent) for any Boogie procedure in addition to using structured Boogie procedure names. When the Boogie result is available and a result handler is associated with the corresponding Boogie procedure, AutoProof will notify the result handler. This mechanism is used for certain verifications that are not directly related to an Eiffel routine like the admissibility check for class invariants.

3.5.7 Extending AutoProof

AutoProof has been built with extendibility in mind. It offers multiple features to extend the translation to Boogie, with or without having to touch the actual implementation.

Boogie mapping

Classes can specify a Boogie file that should be included in the generated Boogie file whenever an entity of this class is used. This allows to include an axiomatization or background theory related to a specific class. In addition, a per class option allows to map an Eiffel type to a Boogie type and a per routine option allows to map Eiffel functions to Boogie functions. This Boogie mapping is used to axiomatize and translate the MML library.

Extension points

AutoProof offers three extension points: *calls*, *nested expressions*, and *across expressions*. Handlers can be registered for each extension point. When the translator hits one of the extension points in the traversal of the Eiffel AST it will call each registered handler in a chain of responsibility style [74]. The

first handler that replies positively will do the translation and if no handler accepts the default translation will continue. The three extension points are:

- The *call* extension point triggers at every routine call. The information provided to the handler are the target and called routine. When the handler does the translation of the call to Boogie it is also its responsibility to translate the arguments of the call, though the handler can delegate the translation of subexpressions like the arguments back to the default translator.
- The *nested expression* extension point triggers at every nested expression. Given the AST node, the handler can inspect the AST surrounding the nested expression to decide if it wants to handle the expression. This extension point is more general than the *call* extension point but harder to use.
- The *across expression* extension point triggers at across expressions (but not at across loop statements). This is an important extension point for AutoProof, as it allows to implement translations for universal and existential quantifiers. The handler will be called for multiple points in the translation, first to set up the quantifier and then for each time the internal cursor is used (e.g. for calls to `item`).

Extension points are used in AutoProof for special translations of built-in features and several base library features. Table 18 gives an overview of implemented extension handlers, and what they are used for.

3.6 Related Work

In reviewing related work, we focus on the tools that are closer to AutoProof in terms of features, design principles, and goals. Only few of them are, like AutoProof, auto-active, work on real object-oriented programming languages, and support the verification of general functional properties. Krakatoa [66] belongs to this category, as it works on Java programs annotated with a variant of JML (the Java Modeling Language [91]). Since it lacks a full-fledged methodology for class invariants and framing, using Krakatoa to verify object-oriented idiomatic patterns—such as those we discuss in Section 4—would be impractical; in fact, the reference examples distributed with Krakatoa target the verification of algorithmic problems where object-oriented features are immaterial. Similar observations apply to the few other auto-active tools working on Java and JML, such as ESC/Java2 [40, 32][4] or

[4]With the `-loopsafe` option which performs sound verification.

Target	Type	Purpose
ANY	call / nested	Special translation for a few routines of class ANY.
TYPE	call / nested	Special translation for a few routines of class TYPE.
NUMERIC	call	Map several functions of numeric types (integers and reals) to Boogie functions from the background theory.
Logical	call / nested	Use functions from the background theory for operations involving MML classes (e.g. equality comparison).
Ownership	call	Implements special handling for ghost features involved in ownership and semantic collaboration.
MML	across	Use MML types in across expressions.
INTERVAL	across	Use integer intervals in across expressions.
SIMPLE_ARRAY	across	Use SIMPLE_ARRAY in across expressions.
SIMPLE_LIST	across	Use SIMPLE_LIST in across expressions.

Table 18: Implemented handlers for extension points.

the more recent OpenJML [39, 123]. Even when ESC/Java2 was used on a few industrial-strength case studies (such as the KOA e-voting system [85]), the emphasis was on modeling and correct-by-construction development, and verification was normally applied only to limited parts of the systems. By contrast, the Spec# system [20, 19] was the forerunner in a new research direction, also followed by AutoProof, that focuses on the complex problems raised by object-oriented structures with sharing, object hierarchies, and collaborative patterns. Spec# works on an annotation-based dialect of the C# language, and supports an ownership model which is suitable for hierarchical object structures; as well as visibility-based invariants to specify more complex object relations. Collaborative object structures as implemented in practice require, however, more flexible methodologies [134] not currently available in Spec#. Tools, such as VeriFast [82], based on separation logic provide powerful methodologies through different abstractions than class invariants, which typically leads to a lower level of automation than tools such as AutoProof and a generally higher annotation overhead—ultimately targeting highly trained users.

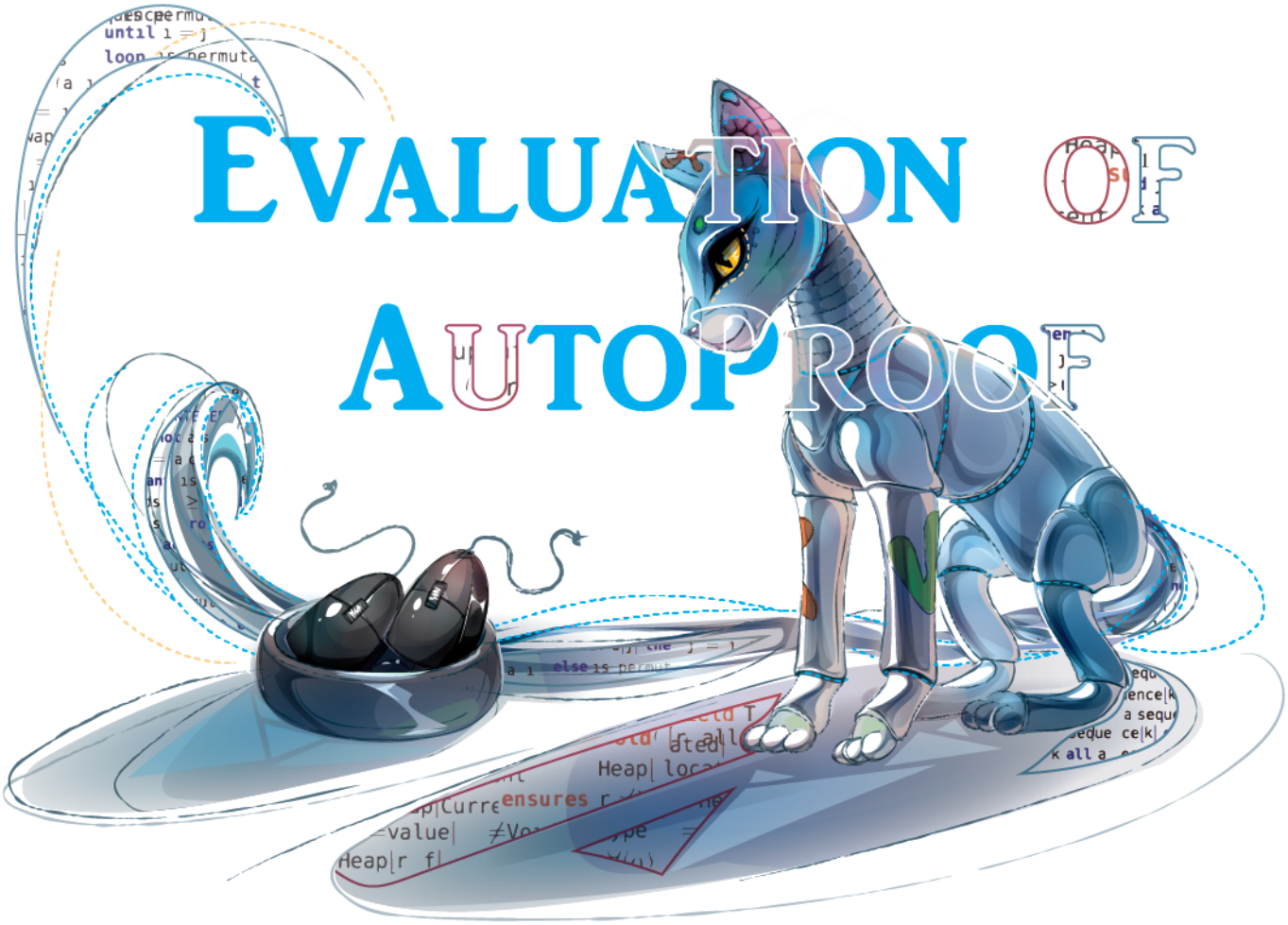
The experience with the Spec# project suggested that targeting a real object-oriented programming language introduces numerous complications

and may divert the focus away from the fundamental problems in tool-supported verification. The Dafny program verifier [94] was developed based on this lesson: it supports a simple language expressly designed for verification, which eschews many of the complications of real object-oriented programming languages (such as inheritance and a complex memory model). Other auto-active verifiers entirely avoid the object-oriented paradigm. For example, Leon [141] and Why3 [27, 67] work on functional programming languages—respectively, a subset of Scala and a dialect of ML; VCC [38] works on C programs and supports object invariants but with an emphasis on memory safety of low-level concurrent code.

AutoProof lies between automatic and interactive tools in the wide spectrum of verification tools. The CodeContract checker (formerly known as Clousot [102]) is a powerful static analyzer for .NET languages that belongs to the former category (and hence it is limited to properties expressible in its abstract domains). The KeY system [22] for Java belongs to the latter category: while it supports SMT solvers as back-ends to automatically discharge simple verification conditions, its full-fledged usage requires explicit user interactions to guide the prover through the verification process.

Chapter 4

Evaluation of AutoProof



After introducing our verifier AutoProof in Chapter 3, this chapter shows how AutoProof can solve challenging verification problems, give an overview of examples verified with AutoProof, and describes the use of AutoProof in a software verification course.

4.1 Verification Challenges

For better or worse, benchmarks shape a field [125]. Patterson's compelling analysis of the coming of age of computer architecture seems to fit the progress of formal software verification too – possibly with a couple-of-decade time shift. As verification techniques left the realm of pure theory and became implementable and usable, they often reported incomparable results: different tools that work on different languages and solve different problems (such as extended static checking, functional correctness, shape analysis, and so on).

Verification competitions and challenges [87, 28, 68, 80] can help in this regard: by providing benchmarks for verification techniques and tools, they help assess progress, compare different approaches, and reward incremental, yet practically relevant, advancements. Hopefully, this will also lead to an outcome similar to computer architecture's: *when a field has good benchmarks, we settle debates and the field makes rapid progress* [125].

This section presents in detail the capabilities of AutoProof to solve several problems from verification competitions. We have selected problems from different verification competitions highlighting various aspects of using AutoProof:

- *Longest Common Prefix (LCP)* [80]: This problem was given at the FM 2012 verification challenge; we use it to discuss various aspects of AutoProof like the use of quantifiers and handling of overflows.

- *Tree Max* [28]: A problem of the COST 2011 competition; its difficulty lies in the need to model the linked structure (using ownership in the case of AutoProof), specifying a global property over the structure, and proving termination of recursion.

- *Sum and Max* [87]: This example from the VSTTE 2010 competition highlights the use of non-linear arithmetic and weak purity.

- *Two-way sort* [68]: With the two-way sort algorithm from the VSTTE 2012 competition we show how inlining of helper routines can significantly reduce the annotation burden and how ghost functions can be used in specifications.

4.1.1 Longest Common Prefix

The first challenge is a *longest common prefix* algorithm: given an array `a` and two indices `x` and `y` within its bounds, determine the length of the

longest common prefix starting at positions `x` and `y` (that is the length of the maximal subarrays from `x` and `y`). Consider, for example, the integer array $\ll 1, 2, 3, 4, 1, 2, 3 \gg$ and the indices[1] 1 and 5 within it: the longest common prefix is the sequence $\ll 1, 2, 3 \gg$ of length 3. For the same array, the longest common prefix for indices 1 and 3 is the empty sequence because the elements at positions 1 and 3 differ; and the longest common prefix for indices 1 and 1 is obviously the whole array (of length 7).

Figure 19 shows an Eiffel implementation of the longest common prefix algorithm as a routine `lcp` fully annotated with precondition (**`require`**), postcondition (**`ensure`**), loop **`invariant`** and loop **`variant`** (also called "ranking function"). The contracts consist of implicitly conjoined assertions; each assertion may have a label, such as `no_overflow` on line 6. AutoProof can automatically verify this implementation against its specification. Specifically, it proves that, for inputs satisfying the precondition, the postcondition holds when the routine terminates, the loop invariant is inductive, the loop terminates, all array accesses are valid, and there are no integer overflows. We now look into these aspects in detail.

Functional Correctness

The postcondition specifies the functional correctness of `lcp` by describing four characterizing properties that the returned integer **`Result`** must satisfy to represent correct output:

- `end_in_range_1`: the output **`Result`** defines a valid subarray at `x`.
- `end_in_range_2`: the output **`Result`** defines a valid subarray at `y`.
- `is_common`: the two subarrays `a`[`x`:`x`+**`Result`**−1] and `a`[`y`:`y`+**`Result`**−1] of length **`Result`** starting at `x` and `y` are pairwise identical. This postcondition uses Eiffel's **`across`**..**`all`** syntax equivalent to the universal quantification $\forall i \in [0..\texttt{Result} - 1] : a[x + i] = a[y + i]$ over the finite integer range $[0..\texttt{Result} - 1]$.
- `longest_prefix`: the two subarrays of length **`Result`** starting at `x` and `y` are maximal; that is, either one of them runs until `a`'s end or the next pair of characters after the subarrays differ. This postcondition uses Eiffel's **`or else`** short-circuited disjunction.

The specification is completed by the loop invariant, which is identical to the first three components of the postcondition, and necessary to establish

[1] We assume arrays numbered from one, as is the norm in Eiffel.

```
1  lcp (a: ARRAY [G]; x, y: INTEGER): INTEGER
2      -- Length of the longest common prefix of a[x..] and a[y..].
3    require
4      x_in_range: 1 ≤ x and x ≤ a.count
5      y_in_range: 1 ≤ y and y ≤ a.count
6      no_overflow: a.count < {INTEGER}.max_value
7    do
8      from Result := 0
9      invariant
10       end_in_range_1: x + Result ≤ a.count + 1
11       end_in_range_2: y + Result ≤ a.count + 1
12       is_common: across 0 |..| (Result−1) as i all a[x+i] = a[y+i] end
13     until
14       x + Result = a.count + 1 or else
15       y + Result = a.count + 1 or else
16       a[x + Result] ≠ a[y + Result]
17     loop Result := Result + 1
18     variant a.count − Result + 1
19     end
20   ensure
21     end_in_range_1: x + Result ≤ a.count + 1
22     end_in_range_2: y + Result ≤ a.count + 1
23     is_common: across 0 |..| (Result−1) as i all a[x+i] = a[y+i] end
24     longest_prefix: (x + Result = a.count + 1) or else
25             (y + Result = a.count + 1) or else
26             (a[x + Result] ≠ a[y + Result])
27   end
```

Figure 19: Implementation of the *lcp* challenge of FM 2012.

them. Postcondition `longest_prefix` follows from the exit conditions on lines 13–16. Notice the peculiar structure of Eiffel loops: the **from** clauses is evaluated once, as if it were regular code appearing before the loop (it is just syntactic sugar); the exit condition in the **until** clause is evaluated before every iteration; correspondingly, the loop body (**loop** clause) may be executed zero times or more.

Our initially unsuccessful attempts at verifying `lcp` prompted us to introduce an improvement in the Boogie translation which is more amenable to automated reasoning with Boogie. The original translation rendered `is_common` roughly as follows:

```
∀ i: int • (0 ≤ i ∧ i ≤ Result−1) ⟹
```

```
(Heap[a, x+i] =Heap[a, y+i])
```

where `Heap` is a mapping representing fields allocated in the heap. Boogie cannot establish that this assertion implies the translation of `is_common`, even if the two assertions are identical in Eiffel (and hence in Boogie). The problem traced back to using the arithmetic operation + to adding a logic variable and a global variable (the problem does not occur when we add a variable to a numeric constant). We solved the problem by introducing Boogie logic functions wrapping arithmetic operations within the scope of quantifiers, such as

```
function add(a, b: int): int { a + b }
```

for addition. The Boogie translation of the loop invariant `is_common` simply becomes

```
∀ i: int • (0 ≤i ∧ i ≤Result−1) ⟹
  (Heap[a, add(x, i)] =Heap[a, add(y, i)])
```

which Boogie can reason about without difficulties.

This trick does not affect the semantics of the translation or what properties can be expressed, but was necessary to accommodate a peculiarity of Boogie's behavior, namely that instantiation triggers cannot include interpreted symbols like the plus sign [96]. This is a recurring scenario for tool developers whose implementations depend on others' tools.

Framing

AutoProof uses sensible defaults for framing specifications. The default for functions (routines with a return value) is *weak purity* [115]. The `lcp` function does not modify any global variables nor allocates new objects and is therefore (strongly) *pure*. AutoProof checks this, and would report violations as verification errors.

Array Accesses

Using the precondition and the loop invariants which restrict the range of the two index variables to always be in the range of the array, AutoProof also verifies that all array accesses are within `a`'s bounds. This entails, in particular, that predicates involving arrays used in the specification are well-formed; the loop's exit condition, for example, evaluates the last disjunct only if the first two evaluate to false (**`or else`** is short-circuited), which implies that `x` + **`Result`** and `y` + **`Result`** are in bounds.

Integer Overflows

AutoProof has an option to verify that no arithmetic operations may overflow. Preconditions `x_in_range` and `y_in_range` specify that `x` and `y` are in bounds, but this is not enough to guarantee that no overflow occurs: the index of the last element of an array with size the largest machine integer `max_value` is the value 1 + `max_value` (array indexing starts at 1 in Eiffel), which produces an overflow. Thus, precondition `no_overflow` restricts the size of the array to less than the maximum integer value. Under this additional precondition, AutoProof verifies that there are no integer overflows.

Termination

AutoProof uses the loop variant on line 18 to prove termination of the loop. It also checks that the variant is a valid variant, that is it decreases after every loop iteration and has a lower bound (determined in this case by the size of the array `a.count`).

Clients

In addition to verifying the *lcp* routine against its specification, we can use AutoProof to check client code that calls *lcp*. For example, the following test cases initialize an array with seven integer values (using the Eiffel syntax $\ll\cdots\gg$), call *lcp* on the array with different values for `x` and `y`, and assert (**check** in Eiffel) that the results are correct.

```
local
  a: ARRAY [INTEGER]
do
  a := ≪1, 2, 3, 4, 1, 2, 3≫
  check lcp (a, 1, 5) = 3 end
  check lcp (a, 2, 6) = 2 end
  check lcp (a, 1, 1) = 7 end
  check lcp (a, 1, 3) = 0 end
end
```

Even if all assertions are logical consequences of `lcp`'s postcondition, the Boogie translation produced by AutoProof fails to verify the last one. We find a quick fix consisting of adding an assertion that explicitly mentions a special fact about the array values:

```
check a[1 + 0] ≠ a[3 + 0] end
```

The translation of this new assertion acts as a trigger to instantiate quantifiers, which Boogie passes on to Z3 and makes verification of the following

assertion succeed. The assertion $\mathsf{a}[1 + 0] \neq \mathsf{a}[3 + 0]$ may be placed at any point in the client before the assertion where the trigger is necessary, since Boogie collects all assertions it has encountered so far. Based on this additional explicit piece of information, Boogie realizes that there are no valid instantiations of the quantifier, and hence the result must be 0.

Since it may be hard for the client to anticipate the need for such an additional assertion, we suggest generalizing it into a postcondition of `lcp`:[2]

$$(\texttt{Result} = 0) = (\mathsf{a}[\mathsf{x}] \neq \mathsf{a}[\mathsf{y}])$$

which does not affect the specification of the routine but makes it more readily usable to verify arbitrary clients. With this postcondition, Boogie also verifies calls to `lcp` that return 0 without the need to suggest quantifier instantiations.

4.1.2 Tree Max

The Tree Max challenge consists of specifying and verifying a function that computes the maximum value of a binary tree. The difficulty in this problem lies in the linked structure of the tree and the need to specify a property over the whole tree. A slightly simplified version of our Eiffel solution is shown in Figure 20. In this solution we model the tree values as a sequence and use the sequence to verify functional correctness of the maximum function and to prove termination of the recursion. The integrity of the tree is guaranteed by using ownership.

Ownership

AutoProof supports a dynamic ownership model. We use ownership to guarantee the tree structure; each parent tree owns the two subtrees `left` and `right`. The ownership relation is defined by the `owns_def` class invariant which specifies the `owns` set (a ghost set that contains all owned objects) of the tree to contain exactly the two subtrees. To be able to claim the two subtrees in the sense of ownership, the constructor that takes the two subtrees needs to guarantee that the arguments are unowned. For this, a precondition is added (lines 20 and 21) that requires the two subtrees be *free* (i.e. not owned). Having ownership of the two subtrees also allows us to use the state of the objects in the class invariant.

[2]The operator = represents both equality and double implication (for Booleans).

```
1  class BINARY_TREE
2
3  feature -- Initialization
4
5    make (v: INTEGER)
6        -- Initialize node.
7      do
8        value := a_value
9        sequence := ≪value≫
10     ensure
11       value_set: value = v
12       no_left: left = Void
13       no_right: right = Void
14     end
15
16   make_children (v: INTEGER;
17             l, r: BINARY_TREE)
18       -- Initialize node.
19     require
20       l.is_free
21       r.is_free
22     do ··· ensure
23       value_set: value = v
24       left_set: left = l
25       right_set: right = r
26     end
27
28 feature -- Access
29
30   value: INTEGER
31       -- Value of this node.
32
33   left, right: BINARY_TREE
34       -- Children nodes.
35 feature -- Basic operations
36
37   maximum: INTEGER
38       -- Maximum value of this tree.
39     require
40       decreases (sequence)
41     do
42       Result := value
43       if left ≠ Void then
44         Result :=
45             Result.max (left.maximum)
46       end
47       if right ≠ Void then
48         Result :=
49             Result.max (right.maximum)
50       end
51     ensure
52       is_max: across sequence.domain as i
53                 all sequence[i] ≤ Result end
54       exists: sequence.has (Result)
55     end
56
57 feature -- Specification
58
59   sequence: MML_SEQUENCE [INTEGER]
60       -- Sequence of values.
61     note ghost end
62
63 invariant
64   owns_def: owns = [left, right]
65   seq_def: sequence = left.sequence +
66       value + right.sequence
67
68 end
```

Figure 20: Implementation of the *tree maximum* challenge of COST 2011.

Tree Model

We model the values of the tree rooted in the **Current** object as a *sequence*. This allows us to express global properties over the tree. The `sequence` attribute (line 59) has the type `MML_SEQUENCE`, a class of the Mathematical Model Library (or MML for short). The sequence is defined in the class invariant `seq_def` to contain the values of the left subtree followed by the value of the **Current** tree and finally the values of the right subtree (for simplicity we ignore the case where `left` or `right` are **Void**). We are allowed to rely on the state of the two subtrees due to the ownership model that

guarantees that all modifications of the owned objects go through the owner.

The sequence is a model of the tree and is declared as *ghost* using Eiffel's annotation mechanism as shown on line 61. We nevertheless need to initialize the value to use it properly, which is done in the creation routines as exemplified on line 9 (the shorthand Eiffel syntax $\ll\cdots\gg$ not only works for arrays but also for MML sets and sequences).

Functional Correctness

The postcondition of `maximum` specifies the properties necessary for functional correctness:

- `is_max`: all values in the model sequence are smaller or equal to the value returned.
- `exists`: the value returned is indeed a value of the sequence.

These two clauses together with the class invariant guarantee that `maximum` returns the maximum value of the binary tree. The correctness of the postcondition follows from the postcondition of the recursive calls.

The `is_max` postcondition can also be written in a simpler way:

```
across sequence as i all i ≤ Result end
```

AutoProof will not verify this assertion however. The iteration over `sequence` directly will iterate over the *range* of the sequence which is a set. An additional assertion is necessary that connects the range of `sequence` with the union of the range of `left.sequence`, `right.sequence`, and the singleton set containing `value`:

```
sequence.range = left.sequence.range + ≪v≫ + right.sequence.range
```

Adding this assertion at the end of `maximum` would allow AutoProof to verify the simpler version of the `is_max` postcondition.

Termination

AutoProof proves termination of the recursion using the decreases annotation on line 40. The decreases clause contains a value or MML expression that has a strict order operator. The value needs to be non-negative or non-empty at the beginning of the routine and strictly smaller when evaluated at the callee site. In this case, the `sequence` of the subtrees are strictly smaller than the `sequence` of the parent tree, as the parent's `sequence` is composed of the sequences of the child nodes together with the singleton sequence containing `value`.

4.1.3 Sum and Max

For the third challenge we are required to verify an algorithm that computes the sum and the maximum element of an array. The specification to be proven is not complete, but only asserts that the maximum value times the array length is an upper bound on the sum. Figure 21 shows an Eiffel implementation with the given specification, which AutoProof can verify; routine `sum_and_max` returns a tuple with the values for sum and max. Two aspects of this algorithm are interesting from the perspective of automated verification.

```
1  sum_and_max (a: ARRAY [INTEGER]): [sum: INTEGER; max: INTEGER]
2      -- Calculate sum and maximum of array 'a'.
3    require
4      a_not_empty: a.count <0
5    local
6      i, sum, max: INTEGER
7    do
8      from
9        i := 1
10     invariant
11       i_in_range: 1 ≤ i and i ≤ a.count + 1
12       sum_in_range: sum ≤ (i−1) * max
13     until
14       i >a.count
15     loop
16       sum := sum + a[i]
17       if a[i] >max then
18         max := a[i]
19       end
20       i := i + 1
21     end
22     Result := [sum, max]
23   ensure
24     sum_in_range: Result.sum ≤ a.count * Result.max
25   end
```

Figure 21: Implementation of the *sum & max* challenge of VSTTE 2010.

Non-linear Arithmetic

The assertions use nonlinear arithmetic (that is, multiplication of variables). Boogie has some support for integer multiplication and division, which AutoProof leverages in the translation to model Eiffel's semantics of integer operations.

Integer Overflows

Enabling overflow checking in AutoProof will reveal several problems in this example. Similar to the *longest common prefix* challenge, the array index `i` may overflow when the array size is equal to the largest possible integer value due to array indexing starting at 1. This can easily be fixed by restricting the array bounds to be strictly smaller then the maximum integer value.

Additionally, AutoProof will report that the summation may overflow. Adding and verifying specification to remedy this is non-trivial. A ghost function expressing array summation is required to add a precondition that the array sum is in the range of integer values.

```
sequence_sum (s: MML_SEQUENCE [INTEGER]): INTEGER
    -- Sum of elements in 's'.
  note functional
  do
    Result := if s.is_empty then 0
              else s.last + seq_sum (s.but_last) end
  end
```

Adding such a ghost function will result in an overflow violation of the ghost function itself. It is therefore not possible to express the property that the sequence sum is bound by a specific integer value with AutoProof while having overflow checking enabled. This suggests that AutoProof should be extended to support partial checking of integer overflows, allowing ghost functions to use mathematical integers and making richer specifications in ghost code possible.

Framing

The second interesting aspect is framing. Line 22 implicitly creates a new tuple[3] to store the result. As mentioned before, functions are by default *weakly pure*, which denotes routines that do not modify the object's state but may allocate and change `fresh` objects. Therefore, we do not need to

[3] `TUPLE` is a reference type in Eiffel.

add any framing annotations to the function, and clients can assume that none of the state accessible by them has changed.

4.1.4 Two-way Sort

Lastly we look at an algorithm that sorts Boolean arrays in linear time. The algorithm scans the input array from both ends, looking for and swapping pairs of inverted elements. It is a technique similar to Dijkstra's Dutch flag algorithm [55] but working on the two Boolean values rather than on the three flag colors. The specification is that the array is sorted and a permutation of the original array when the algorithm terminates. Our implementation is shown in Figure 22.

Functional Correctness

AutoProof will add an implicit precondition to the `two_way_sort` routine that the array `a` is not **`Void`**. The postcondition of the routine specifies the sorted property using two individual postconditions `falses` and `trues`, which denote that the first **`Result`** values of the array are **`False`** and the remaining values are **`True`**. The use of a return value here is only necessary to express this postcondition, hence it should be a ghost value. As it is not possible to express ghost return values in Eiffel we use a normal return value here instead. It would be possible to specify and verify the postcondition without a return value using an existential quantifier (expressed in Eiffel as an **`across`**..**`some`** expression). The reasoning about this is difficult for Boogie though and a successful verification needs to employ the trick of using a helper function as trigger. For this, one declares a function that always returns true, asserts the function with the witness necessary for instantiating the existential quantification, and then uses the function in the existential quantifier that needs to be verified. So by replacing line 24 of `two_way_sort` with

```
check trigger(i) end
```

the two postconditions `trues` and `falses` could be replaced by

```
across 0 |..| a.count as x some
  trigger(x) and
  across 1 |..| x as i all not a.sequence[i] end and
  across x+1 |..| a.count as i all a.sequence[i] end
end
```

thus removing the need of returning a value. The function `trigger` that is necessary for this scheme to work trivially returns **`True`**. Its purpose is only to serve as a trigger for Boogie:

```
two_way_sort (a: ARRAY [BOOLEAN]): INTEGER
    -- Sort boolean array 'a' in linear time.
    -- Returns number of False elements in array.
  note impure
  require modify (a)
  local i, j: INTEGER
  do
    from j := a.count
    invariant
      bounds: i ≥ 0 and i ≤ j and j ≤ a.count
      falses: across 1 |..| i as k all not a.sequence[k] end
      trues: across (j+1) |..| a.count as k all a.sequence[k] end
      is_permutation: is_permutation (a.sequence, old a.sequence)
    until i = j
    loop
      if not a[i+1] then i := i + 1
      elseif a[j] then j := j − 1
      else
        i := i + 1
        swap (a, i, j)
        j := j − 1
      end
    end
    Result := i
  ensure
    falses: across 1 |..| Result as k all not a.sequence[k] end
    trues: across (Result+1) |..| a.count as k all a.sequence[k] end
    is_permutation: is_permutation (a.sequence, old a.sequence)
  end

swap (a: ARRAY [BOOLEAN]; i, j: INTEGER)
    -- Swap elements 'i' and 'j' in array 'a'.
  note inline, skip
  local t: INTEGER
  do t := a[i] ; a[i] := a[j] ; a[j] := t end

is_permutation (s1, s2: MML_SEQUENCE [INTEGER]): BOOLEAN
    -- Are 's1' and 's2' permutations of each other?
  note functional, ghost
  do Result := s1.to_bag ~ s2.to_bag end
```

Figure 22: Implementation of the *two-way sort* challenge of VSTTE 2012.

```
trigger (x: INTEGER) note ghost do Result := True end
```

The last postcondition expressing that the output array is a permutation of the input array relies on the helper function `is_permutation` that we describe below. All postconditions follow trivially from the loop invariant which expresses the progress of starting at both ends of the array and swapping inversed elements.

Ghost Functions

It can be beneficial to encode properties that are used often in helper functions. In the case of `two_way_sort`, we have done this with `is_permutation`, which takes two MML sequences and checks whether they are permutations of each other. The implementation falls back on another MML data type: the *bag* (or multiset). If the bag-representations of the two sequences are identical, then they must be permutations. Since the arguments to the function are MML types, which are only used to represent ghost state like the sequence representation of the array, the function is declared as *ghost* as well (line 39). In addition, the function is also declared *functional*. Functional functions can only contain a single assignment to the result and no additional statements. In the Boogie translation the function is translated to a Boogie **`function`**, making its use in specifications more efficient.

Framing

Functions are by default assumed to be pure. For the implementation of `two_way_sort` of Figure 22, where we sort the array but also return a value, we have to override the default behavior of AutoProof. This can be done by adding the *impure* annotation on line 4 and explicitly specifying that the array will be modified with the *modify* clause on line 5.

Inlining

An interesting aspect which demonstrates AutoProof's capabilities is the usage of a separate routine `swap` to switch inverted elements. Standard verifiers leverage modular reasoning, which entails that the effect of a routine call within the caller is limited to what is mentioned in the callee's specification (its postcondition, in particular). Therefore, verification of implementations such as that in Figure 22 would fail because `swap` has no specification, and hence its effect within `two_way_sort` is undetermined. AutoProof, however, supports two-step verification: after a first unsuccessful attempt at modular verification, it tries to inline `swap`'s body within `two_way_sort` and notices

that verification is successful in this case. AutoProof reports such "partially successful" attempts specially in the GUI, and offers two options: either just use `swap` inlined whenever it is called, or provide a suitable specification to `swap` so that the correctness proof can be carried out modularly. In this simple example, where `swap` is just a helper function and writing its complete specification seems an overkill, we opted for the first option: we added the annotation *`inline`* on line 33, which makes AutoProof inline `swap` whenever necessary without complaining about its lack of specification. This reduces the specification burden on users, thus making the whole verification a bit more practical. In addition, we also added the annotation *`skip`* to tell AutoProof to ignore the `swap` routine entirely when verifying the class, as otherwise AutoProof would complain about the array accesses in `swap`. Skipping inlined routines is not an issue for correctness, as their implementation will be verified as part of the caller.

4.2 Verified Software Repository

To demonstrate AutoProof's competitiveness, we have evaluated AutoProof on a suite of benchmark problems. We only give capsule descriptions of the problems here; the complete solutions are available online through AutoProof's web interface:

`http://se.inf.ethz.ch/research/autoproof/repo`

4.2.1 Benchmarks Description

Our selection of problems is largely based on the verification challenges put forward during several scientific forums, namely the SAVCBS workshops [137], and various verification competitions [87, 28, 68, 80] and benchmarks [156].

Table 23 presents a short description of verified problems. For complete descriptions see the references (and [134] for solutions to problems 11–17). The table is partitioned in three groups: the first group (1–10) includes mainly *algorithmic* problems; the second group (11–17) includes object-oriented design challenges that require complex *invariant* and *framing* methodologies; the third group (18–27) targets *data-structure* related problems that combine algorithmic and invariant-based reasoning. The second and third group include cutting-edge challenges of reasoning about functional properties of objects in the heap; for example, PIP describes a data structure whose node invariants depend on objects not accessible in the physical heap.

#	NAME	DESCRIPTION	FROM
1	Arithmetic (ARITH)	Build arithmetic operations based on the increment operation.	[156]
2	Binary search (BINS)	Binary search on a sorted array (iterative and recursive version).	[156]
3	Sum & max (S&M)	Sum and maximum of an integer array.	[87]
4	Search a list (SEARCH)	Find the index of the first zero element in a linked list of integers.	[87]
5	Two-way max (2-MAX)	Find the maximum element in an array by searching at both ends.	[28]
6	Two-way sort (2-SORT)	Sort a Boolean array in linear time using swaps at both ends.	[68]
7	Dutch flag (DUTCH)	Partition an array in three different regions (specific and general verions).	[55]
8	LCP (LCP)	Longest common prefix starting at given positions x and y in an array.	[80]
9	Rotation (ROT)	Circularly shift a list by k positions (multiple algorithms).	[70]
10	Sorting (SORT)	Sorting of integer arrays (multiple algorithms).	
11	Iterator (ITER)	Multiple iterators over a collection are invalidated when the content changes.	[137, '06]
12	Subject/observer (S/O)	Design pattern: multiple observers cache the content of a subject object.	[137, '07]
13	Composite (CMP)	Design pattern: a tree with consistency between parent and children nodes.	[137, '08]
14	Master clock (MC)	A number of slave clocks are loosely synchronized to a master.	[18]
15	Marriage (MAR)	Person and spouse objects with co-dependent invariants.	[98]
16	Doubly-linked list (DLL)	Linked list whose nodes have links to left and right neighbors.	[98]
17	PIP (PIP)	Graph structure with cycles where each node links to at most one parent.	[140]
18	Closures (CLOSE)	Various applications of function objects.	[92]
19	Strategy (STRAT)	Design pattern: a program's behavior is selected at runtime.	[92]
20	Command (CMD)	Design pattern: encapsulate complete information to execute a command.	[92]
21	Map ADT (MAP)	Generic map ADT with layered data.	[156]
22	Linked queue (QUEUE)	Queue implemented using a linked list.	[156]
23	Tree maximum (TMAX)	Find the maximum value in nodes of a binary tree.	[28]
24	Ring buffer (BUFF)	A bounded queue implemented using a circular array.	[68]
25	Hash set (HSET)	A hash set with mutable elements.	
26	Board game 1 (GAME1)	A simple board game application: players throw dice and move on a board.	
27	Board game 2 (GAME2)	A more complex board game application: different board-square types.	

Table 23: Descriptions of benchmark problems.

#	NAME	#C	#R		CODE	SPECIFICATION								S/C	BOOGIE		TIME [s]
			co	*gh*		*T*	*P*	*Q*	*F*	*L*	*V*	*A*	*C*		*tr*	*bg*	
1	ARITH	1	6	0	99	44	11	12	0	12	9	0	0	0.4	927	579	3.1
2	BINS	1	4	1	62	48	11	12	0	6	3	16	0	1.6	965	1355	3.7
3	S&M	1	1	0	23	12	3	2	1	4	0	2	0	1.0	638	1355	3.9
4	SEARCH	2	5	1	57	62	2	12	2	6	2	27	11	2.3	931	1355	4.1
5	2-MAX	1	1	0	23	12	2	4	0	4	2	0	0	2.3	583	1355	3.0
6	2-SORT	1	2	0	35	28	5	7	2	6	2	6	0	1.8	683	1355	3.2
7	DUTCH	1	4	1	72	75	13	22	4	21	0	15	0	2.6	1447	1355	4.1
8	LCP	2	2	0	40	28	4	7	0	6	2	9	0	1.0	1359	1355	4.2
9	ROT	1	3	3	51	74	14	10	3	17	2	28	0	2.6	1138	1355	4.1
10	SORT	1	9	6	177	219	31	38	9	56	5	80	0	2.6	2302	1355	5.8
11	ITER	3	8	0	88	69	15	26	6	0	0	11	11	1.4	1461	1355	8.9
12	S/O	3	6	0	71	56	10	14	4	3	0	15	10	1.4	1156	1355	4.4
13	CMP	2	5	3	54	125	19	18	5	0	2	72	9	4.3	1327	1355	7.5
14	MC	3	7	0	63	61	9	14	5	0	0	26	7	1.8	956	579	3.7
15	MAR	2	5	0	45	50	12	11	3	0	0	19	5	2.3	755	579	3.3
16	DLL	2	8	0	69	76	12	14	4	0	0	39	7	2.0	891	579	4.4
17	PIP	2	5	1	54	111	23	18	6	0	1	56	7	3.9	988	1355	5.8
18	CLOSE	9	18	0	145	106	40	31	8	0	0	22	5	0.8	2418	688	5.7
19	STRAT	4	4	0	43	5	0	4	0	0	0	1	0	0.2	868	579	3.3
20	CMD	6	8	0	77	32	4	14	2	0	0	10	5	0.7	1334	579	3.3
21	MAP	1	8	0	78	67	6	29	2	6	4	15	5	2.3	1259	1355	4.1
22	QUEUE	4	13	1	121	101	11	26	1	0	0	48	15	1.5	2360	1355	7.4
23	TMAX	1	3	0	31	43	3	12	2	0	2	19	5	2.1	460	1355	3.2
24	BUFF	1	9	0	66	54	8	19	4	0	0	12	11	1.1	1256	1355	4.4
25	HSET	5	14	5	146	341	45	39	10	20	2	197	28	3.7	3546	1355	13.7
26	GAME1	4	8	0	165	93	16	13	4	31	3	10	16	1.2	4044	1355	26.6
27	GAME2	8	18	0	307	173	25	27	11	48	3	29	30	1.4	7037	1355	54.2
	total	72	184	22	2262	2165	354	455	98	246	44	784	184	1.9	43089	1355	203.8

Table 24: Verification of benchmark problems with AutoProof.

4.2.2 Verified Solutions with AutoProof

Table 24 displays data about the verified solutions to the problems of Section 4.2.1; for each problem: the number of Eiffel classes (#C) and routines (#R), the latter split into *gh*ost functions and lemma procedures and *co*ncrete (non-ghost) routines; the lines of executable Eiffel CODE and of Eiffel SPECIFICATION (a total of T specification lines, split into preconditions P, postconditions Q, frame specifications F, loop invariants L and variants V, auxiliary annotations including ghost code A, and class invariants C); the S/C specification to code ratio (measured in tokens)[4]; the lines of BOOGIE input (where *tr* is the problem-specific translation code and *bg* are the included background theory necessary for verification); the overall verification time.

Given that we target full functional verification, our specification to code ratios are small to moderate, which demonstrates that AutoProof's notation

[4]In accordance with common practices in verification competitions, we count *tokens* for the S/C ratio; but we provide other measures in *lines*, which are more naturally understandable.

and methodology support concise and effective annotations for verification. Verification times also tend to be moderate, which demonstrates that AutoProof's translation to Boogie is effective.

To get an idea of the kinds of annotations required, and of their level of abstraction, we computed the ratio A/T of auxiliary to total annotations. On average, 2.8 out of 10 lines of specification are auxiliary annotations; the distribution is quite symmetric around its mean; auxiliary annotations are less than 58% of the specification lines in all problems. Auxiliary annotations tend to be lower level, since they outline intermediate proof goals which are somewhat specific to the way in which the proof is carried out. Thus, the observed range of A/T ratios seems to confirm how AutoProof supports incrementality: complex proofs are possible but require more, lower level annotations.

4.3 AutoProof in the Classroom

AutoProof has been used in a Master's level software verification course at ETH Zurich in the fall semester 2014 [110]. The course introduces students to different techniques and tools of software verification. The students were provided with an introduction and a tutorial for AutoProof, and then used AutoProof for an exercise and a project.

Exercise

In the exercise, students used the web-interface of AutoProof to solve different tasks of increasing difficulty:

1. The students were given a class that implements a simple counter starting from 0 and wrapping around at 59. The class was missing the precondition and several postcondition consisting of basic arithmetic properties, which students had to fill in.

2. Students were given Hoare triples which they had to encode and check in AutoProof.

3. The third task was to add the precondition and loop invariant to an algorithm that computes the maximum of an array. The solution required a first-order loop invariant, which could be adapted from the given postcondition.

4. Given an algorithm that computes the sum and maximum element of an array, the students were asked to provide a loop invariant and postcondition about interesting properties of the algorithm.

5. In the last task, the students were given the implementation of the *longest common prefix* algorithm of the FM 2012 verification competition without any contracts, together with a client of the `lcp` function using it for various test cases. The students were asked to write the full specification of the `lcp` function.

Project

In small teams the students had to use both AutoProof and Boogie [93] to implement and verify a list class. The class uses an array for the storage of elements and has a special sort routine using a different sorting algorithm based on the number and the value range of the list elements. If the list contains more than a predefined number of elements and all these elements are in a given range, the bucket-sort algorithm is used, otherwise quicksort [43]. The two sorting algorithms have non-trivial specifications and, depending on the encoding of the specifications, need additional lemmas to be successfully verified with AutoProof.

Results

There were nine teams that finished the project. The results of using AutoProof are the following:

- Two teams specified and verified the list class completely.
- Five teams achieved very good results and provided full specification, but failed in verifying all components of the sorting routines' postcondition.
- Two teams performed badly, partially due to introducing inconsistencies in the specification without noticing it.

The results indicate that the students were able to successfully use AutoProof to verify a non-trivial algorithm. However, there were two main issues:

1. Teams that achieved good results but failed to completely verify the classes had issues to help AutoProof verify properties of *sets* and *sequences* modelled with MML. Depending on the encoding used to write the specifications, a considerable amount of intermediate assertions or lemmas is necessary to verify such properties.

2. When writing inconsistent specifications, AutoProof will happily verify every program. There are currently no built-in facilities to check if specifications contain a contradiction.

To remedy this situation, students need to be provided with better training on AutoProof. For this, we have created a new tutorial that provides a more systematic approach and more exercises for use in future courses (see Appendix A).

Chapter 5

Methodologies for Auto-Active Verification

This chapter presents methodologies for auto-active verification of programs for both improving usability and extending support for the object-oriented paradigm. All code examples are in Eiffel, with few occasional notational simplifications; even readers not familiar with the language should find the code easy to understand. Extensions to the base Eiffel language use special keywords in *italics*. We use Boogie to demonstrate encodings to an intermediate verification language. For this, we present a short introduction to Boogie in the next section.

5.1 A Short Introduction to Boogie

Boogie is a language for verification [93], as well as an automated verifier that takes programs written in the Boogie language as input. AutoProof verifies Eiffel programs by translating them into the Boogie language and then by calling the Boogie verifier on the translation. The Boogie verifier generates verification conditions for the input program, and supports different prover back-ends (e.g., Z3 and Simplify) to discharge them. For readers unfamiliar with Boogie, this section describes the essential features of the Boogie language used in the rest of the paper.

The Boogie language offers two kinds of constructs: a simple imperative modular programming language—used to translate the source program (Eiffel, in our case)—and a specification language based on first-order logic—used to define specification elements and background logic theories needed to support complex specifications.

Boogie's specification language is a typed first-order logic with arithmetic. The basic types include Booleans (`bool`) and mathematical (unbounded) integers (`int`); the type constructors support the definition of derived types. Line 1 in Figure 25 declares a new type `person`, which Boogie treats as a fresh sort for variables. The specification language supports the definition of global variables and constants, functions (in the sense of mathematical logic), and axioms. Line 2, for example, declares a global constant `eve` of type `person`. Lines 3 and 4 declare two functions `age` and `can_vote`. Lines 5 and 6 introduce two axioms about the declared items: `age` is defined as 23 for argument `eve`; and `can_vote` is true precisely for persons whose age is greater than or equal to 18.

```
1 type person;
2 const eve: person;
3 function age(p: person) returns (int);
4 function can_vote(p: person) returns (bool);
5 axiom (age(eve) = 23);
6 axiom (∀ p: person • can_vote(p) ⟺ age(p) >= 18);
```

Figure 25: Some definitions in Boogie's specification language.

Boogie's programming language supports the definition of procedures. Each procedure has a signature, which may include a specification in terms of preconditions (**`requires`**), postconditions (**`ensures`**), and frame clauses (**`modifies`**). The specification clauses contain formulas in Boogie's specification language. Postconditions, in addition, supports the usage of the **`old`**

keyword to evaluate expressions in the state *before* a procedure was called. Modifies clauses, instead, define a procedure's *frame*, that is the set of global variables the procedure may modify. Pre- and postconditions may be marked as **free**, which prevents the generation of proof obligations based on them: a **free** assertion is assumed to hold whenever convenient, but need not be checked when required.

Procedure implementations use standard imperative constructs (assignments, conditionals, loops, jumps, and procedure calls) with the usual semantics. To write nondeterministic programs, Boogie's programming language includes the **havoc** command, which assigns a nondeterministically chosen value to its argument variables. To constrain the effects of **havoc** and to express intermediate verification conditions, Boogie's programming language also offers **assert** and **assume** statements. Both take an arbitrary formula `F` as argument. The program state of every execution reaching an **assert** `F` must satisfy `F`; otherwise, verification fails. Conversely, the verification process can assume that `F` holds of the program state whenever an execution reaches **assume** `F`, which "shapes" the nondeterministic behavior when convenient.

Figure 26 shows the specification and implementation of a procedure `vote` to cast a vote, demonstrating Boogie syntax.

```
1  var votes: int;
2
3  procedure vote(p: person);
4    requires can_vote(p);
5    ensures votes = old(votes) + 1;
6    modifies votes;
7
8  implementation vote(p: person) {
9    votes := votes + 1
10 }
```

Figure 26: A Boogie procedure `vote`: specification and implementation.

5.2 Overview and Examples

Binary search is a widely-known algorithm and is considered a standard benchmark for software verification [156]. Most programmers have implemented it at least once in their life. According to Knuth [88, Vol. 3, Sec. 6.2.1], their implementations were often "wrong the first few times they tried".

```
1  binary_search (a: ARRAY [INTEGER]; x: INTEGER): INTEGER
2    require a ≠ Void
3    local middle: INTEGER
4    do
5      if a.count = 0 then
6        Result := −1
7      else
8        middle := (1 + a.count) / 2
9        if a[middle] = x then
10         Result := middle
11       elseif a[middle] > x then
12         Result := binary_search (a[1:middle − 1], x)
13       else
14         assert a[middle] < x
15         Result := binary_search (a[middle + 1:a.count]), x)
16         if Result ≠ −1 then Result := Result + middle end
17       end
18     end
19   ensure Result = −1 or (1 ≤ Result and Result ≤ a.count)
20   end
```

Figure 27: An implementation of binary search.

To demonstrate, consider the binary search implementation in Figure 27, which takes an integer array `a` and an integer value `x` and returns an integer index (the value assigned to **`Result`**) pointing to an occurrence of `x` in `a`. Since we assume arrays numbered from one, if `x` is not found in `a` the routine by convention returns -1. The implementation in Figure 27 is indicative of what programmers typically write [130, 61] when using a language supporting specifications in the form of contracts (pre- and postconditions, and intermediate assertions such as loop invariants and **`assert`** instructions): the implementation is "almost" correct (if you do not immediately see the error, read on), and the specification is obviously incomplete.

Part of the missing specification is implicit in the semantics of the programming language, which is probably why the programmer did not bother writing it down explicitly. In particular, arithmetic operations should not overflow for the program to have a well-defined semantics. The midpoint calculation on line 8 overflows when the array size `a.count` has value equal to the largest representable machine integer, even if the value of `middle` is within the bounds; this is indeed a common error in real implementations of binary search [26].

If we try to verify the program in Figure 27 using static verifiers such as Dafny [94], we do not find any error because integer variables are modeled using mathematical integers which do not overflow. In Section 5.3 we discuss our approach which automatically instantiates implicit specification elements that represent tacit assumptions about the programming language semantics. Such *implicit contracts* help early error detection of subtle errors not explicitly specified, such as the potential overflow just discussed. In addition, they are made available within a more general static verification mechanism, where they can complement programmer-written contracts to improve the efficiency of the overall verification process without sacrificing precision.

```
1  two_way_sort (a: ARRAY [BOOLEAN])
2    require a.count >0
3    local i, j: INTEGER
4    do
5      i := 1 ; j := a.count
6      until i ≥ j
7      invariant 1 ≤ i and i ≤ j + 1 and j ≤ a.count
8      loop
9        if not a[i] then
10         i := i + 1
11       elseif a[j] then
12         j := j − 1
13       else
14         swap (a, i, j)
15         i := i + 1
16         j := j − 1
17       end
18     variant j − i + 1
19     end
20   end
21
22 swap (b: ARRAY [BOOLEAN]; x, y: INTEGER)
23   local t: BOOLEAN
24   do
25     t := b[x] ; b[x] := b[y] ; b[y] := t
26   end
```

Figure 28: An implementation of two-way sort of Boolean arrays.

Not only do incomplete specifications limit the kinds of error that can be detected automatically during verification; they may also prevent verifying perfectly *correct* programs as we now illustrate with the example of Figure 28, taken from the VSTTE 2012 verification competition [68]. Routine `two_way_sort` sorts an array `a` of Boolean values in linear time with a technique similar to the partitioning algorithm used in Quick Sort. Two pointers `i` and `j` scan the array from its opposite ends; whenever they point to an inversion (that is, a **`False`** in the right-hand side and a **`True`** in the left-hand side) they remove it by swapping the elements pointed. When the whole array is scanned, it is sorted.

The sorting algorithm calls an auxiliary routine `swap` that exchanges elements; `swap` does not have any specification—again, a situation representative of how programmers typically specify their programs. This is a problem because static verification uses specifications to reason *modularly* about routine calls: the effects of the call to `swap` on line 14 are limited to `swap`'s postcondition. Since it does not have any, the proof of `two_way_sort` does not go through; in particular, it cannot establish that the loop invariant at line 7 is inductive, which would then be the basis to establish the variant as well as any programmer-written postcondition.

In our approach, when modular verification fails the verifier makes another attempt after *inlining* routine bodies at their call sites. As we describe in Section 5.4, the application uses simple heuristics to avoid combinatorial explosion (for example, in the case of recursive calls). With inlining, we can prove that the invariant at line 7 is inductive without need for more specification.

Using inlining, we can also check interesting properties about *clients* of `two_way_sort`. For example, when calling the routine on an empty array, we compare a failed modular verification attempt (which reports a violation of `two_way_sort`'s precondition) to a successful verification with inlining (which only evaluates the routine's body); the discrepancy *suggests* that `two_way_sort`'s precondition is unnecessarily strong and can be relaxed to `a` $\neq$ **`Void`** without affecting the rest of the verification process. This kind of improved feedback, concocted from two different verification attempts, is the two-step verification we present in detail in Section 5.5.

5.3 Implicit Contracts

Implicit contracts are simple specification elements that are implicit in the semantics of the programming language—Eiffel in our examples. Since they are implicit, programmers tend to reason informally about the program without

writing them down as assertions. This limits the kinds of properties that can be proved automatically with a static verifier. With implicit contracts, the verifier transparently annotates the program under verification so that the feedback to users is more accurate and goes deeper than what would have been possible based on the explicitly written contract only. We currently support the following classes of implicit contracts.

5.3.1 Targets Non-Void

A qualified call $t.r_1(a_1).\ldots.r_n(a_n)$, with $n \geq 0$, is non-Void if `t` $\neq$ **`Void`** and, for $1 \leq k < n$, $t.r_1(a_1).\ldots.r_k(a_k)$ returns a non-Void reference. For every such qualified call appearing as instructions or in expressions, we introduce the corresponding implicit contract that asserts that the call is non-Void.

For example, `two_way_sort`'s precondition (Figure 28) is augmented with the implicit contract that `a` $\neq$ **`Void`** following from the qualified call `a.count`.

5.3.2 Routine Calls in Contracts

In programming languages supporting contracts there need not be a sharp distinction between functions used in the implementation and functions used in the specification. Routine `two_way_sort`, for example, uses the function call `a.count`—returning the length of array `a`—in its precondition and loop invariant, but also in the assignment instruction on line 5. Functions used in contracts may have preconditions too; programmers should make them explicit by replicating them whenever the function is mentioned, but they often neglect doing so because it is something that is implicit when those functions are used in normal instructions, whereas it is not checked when the same functions are used in contracts.

Consider, for instance, a function `is_sorted` with the obvious semantics, and suppose that its precondition requires that it is applied to non-empty lists. If `is_sorted` is called anywhere in the implementation, then it is the caller's responsibility to establish its precondition; the caller is aware of the obligation explicit in `is_sorted`'s contract. But if `is_sorted` is called, say, as precondition of `binary_search`, establishing `is_sorted`'s precondition is now the responsibility of callers to `binary_search`, who are, however, unaware of the non-emptiness requirement *implicit* in `binary_search`'s precondition. In fact, the requirement should explicitly feature as one of `binary_search`'s preconditions.

To handle such scenarios automatically, for every call to any function `f` appearing in contracts, we introduce the corresponding implicit contract that asserts that `f`'s precondition holds right before `f` is evaluated in the

contract. If `f`'s precondition includes calls to other functions, we follow the transitive closure of the preconditions, also checking well-formedness (that is, no circularity occurs).

5.3.3 Arithmetic Expressions

The subexpressions $\mathsf{sub}(e)$ of an integer expression e are defined in the obvious way: if e is an integer constant or an integer variable then $\mathsf{sub}(e) = \{e\}$; if e is the application of a unary operator $\sim$, that is $e = \sim d$, then $\mathsf{sub}(e) = \{e\} \cup \mathsf{sub}(d)$; if e is the application of a binary operator $\oplus$, that is $e = c \oplus d$, then $\mathsf{sub}(e) = \{e\} \cup \mathsf{sub}(c) \cup \mathsf{sub}(d)$. For every integer expression e appearing in instructions or expressions, we introduce the implicit contract that asserts that no subexpression of e's may overflow:

$$\bigwedge_{x \in \mathsf{sub}(e)} \{\texttt{INTEGER}\}.\texttt{min\_value} \leq \texttt{x} \quad \textbf{and} \quad \texttt{x} \leq \{\texttt{INTEGER}\}.\texttt{max\_value}$$

For every subexpression of the form $c \oslash d$, where $\oslash$ is some form of integer division, we also introduce the implicit contract $\texttt{d} \neq 0$, which forbids division by zero.

The integer expression at line 8 in Figure 27, for example, determines the implicit contract $1 + \texttt{a.count} \leq \{\texttt{INTEGER}\}.\texttt{max\_value}$, which may not hold.

5.3.4 Related Work

Auto-active verification relies on accurate specifications, which are not easy to write and get right. One way to ameliorate this situation is inferring specifications automatically using static [33, 89, 71] or dynamic [59, 153, 152] techniques. Specifications dynamically inferred are based on a finite number of executions, and hence may be unsound; this makes them unsuitable for use in the context of static verification. Static techniques can infer sound specifications from the program text; these are useful to document existing implementations, to discover auxiliary assertions (such as loop invariants), or for comparison with specifications independently written, but proving an implementation correct against a specification inferred from it is mostly a vacuous exercise.

The simple implicit contracts that we use in our approach express well-formedness properties of the input program, which are tacitly assumed by programmers reasoning informally about it; therefore, there is no risk of circularity. Some static verifiers use mathematical integers or assume purity of specification functions to have well-formedness by construction; a risk is that, when they are applied to real programming languages, the corresponding semantic gap may leave some errors go unnoticed. ESC/Java2 [40], for

example, does not check for overflows [86], nor if specification expressions are executable (for example, null-dereferencing could happen when evaluating a precondition). The Dafny verifier [94] checks well-formedness of pre- and postconditions, and may consequently require users to add explicit contracts to satisfy well-formedness. Our implicit contracts are instead added and checked automatically, without requiring users to explicitly write them. In this sense, they are similar to approaches such as VCC [38], which models the semantics of the C programming language as precisely as possible.

5.4 Inlining and Unrolling

Inlining and unrolling are routinely used by compilers to optimize the generated code for speed; they are also occasionally used for program checking, as we discuss in Section 5.4.3. The novelty of our approach is the automatic combination, in two-step verification, of inlining and unrolling with modular "specification-based" verification. Inlining and unrolling may succeed in situations where little programmer-written specification is available; in such cases, users get a summary feedback that combines the output of each individual technique and is aware of the potential unsoundness of inlining and unrolling. The combined feedback gives specific suggestions as to what should be improved. This section presents the definitions of inlining and unrolling; Section 5.5 discusses how they are combined in two-step verification.

5.4.1 Inlining

The standard approach to reasoning about routine calls is *modular* based on specifications: the effects of a call to some routine `r` within the callee are postulated to coincide with whatever `r`'s specification is. More precisely, the callee should establish that `r`'s precondition holds in the calling context; and can consequently assume that `r`'s postcondition holds and that the call does not modify anything outside of `r`'s declared frame.

The modular approach is necessary to scale verification to large pieces of code. At the same time, it places a considerable burden on programmers, since every shortcoming in the specifications they provide may seriously hinder what can be proved about their programs. This is a practical issue especially for helper functions that are not part of the public API: programmers may not feel compelled to provide accurate specifications—postconditions, in particular—for them because they need not be documented to clients; but they would still like to benefit from automated program checking. This is the case of routine `swap` in Figure 28, which is not specified but whose semantics

is obvious to every competent programmer.

Inlining can help in these situations by replacing abstract reasoning based on specifications with concrete reasoning based on implementations whenever the former are insufficient or unsatisfactory. In particular, inlining is likely to be useful whenever the inlined routine has no postcondition, and hence its effects within the callee are undefined under modular reasoning. Of course, inlining has scalability limits; that is why we apply it in limited contexts and combine it with standard modular verification as we discuss in Section 5.5.

Definition of inlining

Consider a routine `r` of class `C` with arguments `a`, which we represent as:

`r` (`t`: `C` ; `a`) **`require`** P_r *`modify`* F_r **`do`** B_r **`ensure`** Q_r **`end`**

For $n \geq 0$, the n-inlining $\mathsf{inline}(A, n)$ of calls to r in a piece of code A is defined as A with every call `u.r(b)` on target `u` (possibly **`Current`**) with actual arguments `b` modified as follows:

$$\mathsf{inline}(\mathsf{u.r(b)}, n) = \begin{cases} \textbf{assert}\ P_r[u,b];\ \textbf{havoc}\ F_r[u,b];\ \textbf{assume}\ Q_r[u,b] & \text{if } n = 0 \\ \mathsf{inline}(B_r[u,b], n-1) & \text{if } n > 0 \end{cases}$$

Inlining works recursively on calls to routines other than r and recursive calls to r in B_r; non-call instructions are instead unchanged. 0-inlining coincides with the usual modular semantics of calls based on specifications. Otherwise, inlining replaces calls to r with $B_r[u, b]$ (r's body applied to the actual target u and arguments b of the calls), recursively for as many times as the recursion depth n.

Since inlining discards the inlined routine's precondition, it may produce under- or over-approximations of the calls under modular semantics, respectively if the declared precondition is weaker or stronger than the body's weakest precondition.

For any $n > 0$, the n-inlining of `swap` in `two_way_sort`'s body (Figure 28) consists of replacing the call to `swap` at line 14 with `swap`'s body instantiated in the correct context, that is `t` := `a[i]` ; `a[i]` := `a[j]` ; `a[j]` := `t` with `t` a fresh local variable declared inside `two_way_sort`.

Inlining and dynamic dispatching

In programming languages with dynamic dispatching, the binding of routine bodies to routine calls occurs at runtime, based on the dynamic type of the call targets. This is not a problem for modular reasoning because it can rely on behavioral subtyping and the rule that routine redefinitions in descendants

(overriding) may only weaken preconditions and strengthen postconditions. Inlining, instead, has to deal with dynamic dispatching explicitly: in general, verification using inlining of a routine `r` of class `C` is repeated for every overriding of `r` in `C`'s descendants. This also requires to re-verify the system whenever new descendants of `C` are added, unless overriding `r` is eventually forbidden (**`frozen`** in Eiffel, **final** or **private** in Java, or **sealed** in C#). These limitations are, however, not a problem in practice when we apply inlining not indiscriminately but only in limited contexts for small helper routines, and we combine its results with classic modular reasoning as we do in two-step verification.

5.4.2 Unrolling

The standard approach to modular reasoning also applies to loops based on their loop invariants: the effects of executing a loop on the state of the program after it are postulated to coincide with the loop invariant. The inductiveness of the invariant is established separately for a generic iteration of the loop, and so is the requirement that the invariant hold upon loop entry.

This reliance on expressive loop invariants is at odds with the aversion programmer typically have at writing them. This is not only a matter of habits, but also derives from the fact that loop invariants are often complex specification elements compared to pre- and postconditions [71]; and, unlike pre- and postconditions which constitute useful documentation for clients of the routine, loop invariants are considered merely a means to the end of proving a program correct. The loop of `two_way_sort` in Figure 28, for example, has a simple loop invariant that only bounds the values of the indexes `i` and `j`; this prevents proving any complex postcondition.

Unrolling can help in these situations by evaluating the effects of a loop in terms of its concrete body rather than its invariant. This may help prove the postcondition when the invariant is too weak, showing that a certain number of repetitions of the body are sufficient to establish the postcondition. Furthermore, in the cases where we have a way to establish a bound on the number of loop iterations, unrolling precisely renders the implementation semantics. We will generalize these observations when discussing how unrolling is applied automatically in the context of two-step verification (Section 5.5).

Definition of unrolling

Consider a generic annotated loop L:

```
until exit invariant I loop B variant V end
```

which repeats the body `B` until the exit condition `exit` holds, and is annotated with invariant `I` and variant `V`. For $n \geq 0$, the n-unrolling $\mathsf{unroll}(L, n)$ of L is defined as:

$$\mathsf{unroll}(L, n) \;=\; (\textbf{if not } \mathtt{exit} \textbf{ then } \mathtt{B} \textbf{ end})^n$$

where the nth exponent denotes n repetitions. Since unrolling ignores the loop invariant, it may produce under- or over-approximations of the loop's modular semantics, respectively if the declared loop invariant is weaker or stronger than the body's weakest precondition.

5.4.3 Related Work

Inlining and unrolling are standard techniques in compiler construction. The Boogie verifier [93] also supports inlining of procedures: through annotations, one can require to inline a procedure to a given depth using different sound or unsound definitions. Boogie also supports (unsound) loop unrolling on request. AutoProof's current implementation of inlining and unrolling works at source code level, rather than using Boogie's similar features, to have greater flexibility in how inlining and unrolling are defined and used. Methods specified using the Java Modeling Language (JML) with the "helper" modifier [40] are meant to be used privately; ESC/Java inlines calls to such methods [69]. ESC/Java also unrolls loops a fixed amount of times; users can choose between performing sound or unsound variants of the unrolling. Unrolling and inlining can also be used to check the type correctness of JavaScript programs [119]. In two-step verification, we use inlining and unrolling completely automatically: users need not be aware of them to benefit from an improved feedback that narrows down the sources of failed verification attempts.

5.5 Two-Step Verification

Two-step verification builds on the methods presented before, combining them to produce improved user feedback.

Implicit contracts are simply added whenever appropriate and used to have early detection of errors violating them. In AutoProof, which translates Eiffel to Boogie to perform static proofs, implicit contracts are not added to the Eiffel code but are silently injected into the Boogie translation, so that the input code does not become polluted by many small assertions; users familiar with Eiffel's semantics are aware of them without explicitly writing them down. Errors consisting of violations to implicit contracts reference

back the original statements in Eiffel code from which they originated, so that the error report is understandable without looking at the Boogie translation.

Whenever the verifier checks a routine that contains routine calls, two-step verification applies inlining as described in Section 5.5.1. Whenever it checks a routine that contains loops, two-step verification applies unrolling as described in Section 5.5.2. The application of the two steps is completely automatic, and is combined for routines that includes both calls and loops; users only get a final improved error report in the form of suggestions that narrow down the possible causes of failed verification more precisely than in standard approaches. Section 5.5.4 briefly illustrates two-step verification on the running example of Section 5.2.

5.5.1 With Inlining

Consider a generic routine `r` with precondition P_r and postcondition Q_r, whose body B_r contains a call `t.s(a)` to another routine `s` with precondition P_s, postcondition Q_s and body B_s (as shown in Figure 29). Two-step verification runs two verification attempts on r:

1. **Modular verification:** The first step of two-step verification for r follows the standard modular verification approach: it tries to verify that r is correct with respect to its specification, using s's specification only to reason about the call to s; and then it separately tries to verify s against its own specification.

2. **Inlined verification:** The second step of two-step verification for r replaces the call to s in r with $\mathsf{inline}(t.s(a), n)$, for some $n > 0$ picked as explained in Section 5.5.3, and then verifies r with this inlining.

step 1: modular		**step 2: inlined**	
VERIFY r	VERIFY s	VERIFY r	SUGGESTION
P_s fails	success	success	weaken P_s or use inlined `s`
Q_r fails	success	success	strengthen Q_s or use inlined `s`
success	Q_s fails	success	strengthen P_s or weaken Q_s

Table 30: Two-step verification with inlining: summary of suggestions.

Each of the two steps may fail or succeed. According to the combined outcome, we report a different suggestion to users, as summarized in Table 30.

```
r
  require P_r
  do
    B_r { ⋮
          t.s(a)
          ⋮
  ensure Q_r

s
  require P_s
  do
    B_s
  ensure Q_s

q
  require P_q
  do
    ⋮
    L { until e
        invariant I
        loop B
        variant V end
    ⋮
  ensure Q_q
```

Figure 29: A routine `r` calling another routine `s`; and a routine `q` with a loop.

Precondition fails

If modular verification (first step) fails to establish that s's precondition P_s holds right before the call, but both modular verification of s and inlined verification (second step) of r succeed, it means that s's precondition may be inadequate[1] while its implementation is correct with respect to its specification and to the usage made within r. In this case, there are two options: if s is a helper function used only in r or inside a single class, we may as well drop s's specification and just use it inlined wherever needed during verification. In more general cases, we should try to *weaken* P_s in a way that better characterizes the actual requirements of how s is used.

Postcondition fails

If modular verification fails to establish that r's postcondition Q_r holds when B_r terminates, but both modular verification of s and inlined verification of r succeed, it means that s's postcondition fails to characterize r's requirements while s's implementation is correct with respect to its specification. As in the previous case, there are two options: we may drop s's specification and just use it inlined; or we should try to *strengthen* Q_s in a way that better characterizes the actual expectations of r on s. A similar scheme applies not just to failed postconditions Q_r but whenever modular verification fails to verify intermediate assertions occurring on paths after the call to s in r.

[1] As with all failed static verification attempts, we cannot exclude that failure is simply due to limitations of the theorem prover.

Local proof fails

If modular verification fails to establish that s's postcondition Q_s holds when B_s terminates, but both modular verification of r and inlined verification of r succeed, it means that s's specification cannot be proved consistent with its implementation, while the latter is correct with respect to the usage made within r. The suggestion is to change s specification in a way that still accommodates its usage within r and can be verified: strengthen the precondition P_s, weaken the postcondition Q_s, or both. With this information, there is no way to decide if the problem is with the pre- or postcondition, but we can always try to modify either one and run verification again to see if the suggestion changes.

Other cases

In the remaining cases, two-step verification is inconclusive in the sense that it gives the same feedback as modular verification alone. In particular, when the second step fails it is of no help to determine whether the problem is in the specification or the implementation. If, for example, both modular and inlined verification of r fail to establish the postcondition Q_r, but modular verification of s succeeds, we cannot conclude that s's implementation is wrong because it does not achieve Q_r: it may as well be that r's implementation is wrong, or r's postcondition is unreasonable; which is exactly the information carried by a failed modular verification attempt.

Also notice that inlined verification cannot fail when modular verification fully succeeds: if s's implementation satisfies its specification, and that specification is sufficient to prove r correct, then the semantics of s within r is sufficient to prove the latter correct. Therefore, we need not run the second step when the first one is successful.[2]

5.5.2 With Unrolling

Consider a generic routine q with precondition P_q and postcondition Q_q, whose body B_q contains a loop L with exit condition e, invariant I, variant V, and body B (as shown in Figure 29). Two-step verification runs two verification attempts on r:

1. **Modular verification:** The first step of two-step verification for r fol-

[2]Again, exceptions might occur due to shortcomings of the theorem prover used by the modular verifier, which might be able to prove a set of verification conditions but fail on a syntactically different but semantically equivalent set due to heuristics or limitations of the implementation. These are, however, orthogonal concerns.

lows the standard modular verification approach: it tries to verify that r is correct with respect to its specification, using the loop invariant I only to reason about the effect of L within r; and then it separately tries to verify that I is a correct inductive loop invariant (that is, it holds on entry and is maintained by iterations of the loop).

2. **Unrolled verification:** The second step of two-step verification for r replaces the loop L in r with $\mathsf{unroll}(L, n)$, for some $n > 0$ picked as explained in Section 5.5.3, and then verifies r with this unrolling and the additional assertion `assert V ≤ n`, evaluated upon loop entry, that the loop executes at most n times.[3]

step 1: modular	**step 2: unrolled**		
VERIFY r	`assert V ≤ n`	VERIFY r	SUGGESTION
Q_q fails	success	success	use inlined L
Q_q fails	fail	success	strengthen I to generalize
I fails inductiveness	success	success	use inlined L
I fails inductiveness	fail	success	change I to generalize

Table 31: Two-step verification with unrolling: summary of suggestions.

Each of the two steps may fail or succeed. According to the combined outcome, we report a different suggestion to users, as summarized in Table 31.

Postcondition fails

Suppose that modular verification (first step) fails to establish that r's postcondition Q_q holds when B_q terminates, but unrolled verification (second step) of r succeeds. The suggestion in this case depends on whether the prover can also establish the intermediate assertion `assert V ≤ n`. If it does, n is a finite upper bound on the number of loop iterations in every execution. Thus, the loop implementation is correct but the loop invariant I is inadequate to prove the postcondition; we may as well drop the invariant I and just use the exhaustively unrolled loop during verification. In the more general case where the assertion `V ≤ n` fails, the successful unrolled proof shows that the loop body works with a finite number of iterations, and hence it is likely correct; we may then try to strengthen (or otherwise change) the invariant I in a way that captures a generic number of loop iterations and is sufficiently strong to establish Q_q. A similar scheme applies not just to

[3] If a variant is not available or cannot be verified to be a valid variant, we proceed as if the assertion did not hold.

failed postconditions Q_q but whenever modular verification fails to verify intermediate assertions occurring on paths after the loop L in q.

Invariant fails

Suppose that modular verification fails to establish that I is inductive, but unrolled verification of r succeeds. The suggestion depends on whether the prover can also establish the intermediate assertion `assert V ≤ n`. If it does, the loop implementation is correct but the loop invariant I is inadequate; we may as well drop the invariant I and just use the exhaustively unrolled loop during verification. In the more general case where the assertion `V ≤ n` fails, the successful unrolled proof shows that the loop body works with a finite number of iterations, and hence it is likely correct; we may then try to change the invariant I in a way that captures a generic number of loop iterations and is sufficiently strong to establish Q_q. With this information, there is no way to decide if the invariant should be strengthened or weakened, but we can always try either one and run verification again.

In the remaining cases, two-step verification gives the same feedback as modular verification alone. And, as for inlining, we need not run the second step (unrolled verification) when modular verification is completely successful.

5.5.3 Bounds for Nesting and Loops

The application of inlining and unrolling requires a parameter n: the maximum depth of nested calls in the former case; and the number of explicit iterations of the loop in the latter. The choice of n is more subtle for unrolling—where it should represent a number of iterations sufficient to make the second step of verification succeed—than for inlining—where it becomes relevant only in the presence of nested calls or recursion.

We use simple heuristics to pick values for n that are "feasible", that is do not incur combinatorial explosion. In the case of inlining, we get a crude estimate of the size of the inlined program as follows. For a routine p, let π_p denote the total number of simple paths in p from entry to exit. If p has size $|p|$ (measured in number of instructions) and includes m calls to routines $r_1, \ldots, r_m$, we recursively define $\|p\|_n$ as: $|p|$ if $n = 0$, $|p| + \pi_p(\|r_1\|_{n-1} + \cdots + \|r_m\|_{n-1})$ if $n > 0$. The value of $\|p\|$ is an upper bound to the total number of instructions on all possible paths of the inlined routine. Inlining in p uses an n such that $\|p\|_n \leq 10^4$.

In the case of unrolling a loop within routine q, our implementation does some simple static analysis to determine if the calling context of q or q's

precondition suggest a finite bound of the loop (in practice, this is restricted to loops over arrays that are declared statically or with a constant upper bound in the precondition). In such cases, n is simply the inferred bound. Otherwise, we roughly estimate the size of an unrolled loop L as $n|L|$, where $|L|$ is the size of L in number of instructions; unrolling L uses an n such that $n|L| \leq 10^3$.

In many practical cases (in the absence of recursion or deeply nested calls), very small n's (such as $1 \leq n \leq 5$) are sufficient to produce a meaningful results in two-step verification.

5.5.4 Examples

Let us demonstrate how two-step verification works on the examples introduced in Section 5.2. Figure 32 shows two clients of routine `two_way_sort` (Figure 28). Routine `client_1` calls `two_way_sort` on an empty array, which is forbidden by `two_way_sort`'s precondition. Normally, this is blamed on `client_1`; with two-step verification, however, the second verification attempt inlines `two_way_sort` within `client_1` and successfully verifies it. This suggests that `client_1` is not to blame, because `two_way_sort`'s precondition is unnecessarily strong (first case in Table 30), which is exactly what AutoProof will suggest in this case as shown in Figure 33. In fact, the sorting implementation also works on empty arrays, where it simply does not do anything.

```
client_1
  local a: ARRAY [BOOLEAN]
  do
    a := ≪≫ -- empty array
    two_way_sort (a)
  end
```

```
client_2
  local a: ARRAY [BOOLEAN]
  do
    a := ≪True, False, False, True≫
    two_way_sort (a)
    assert a[1] = False
  end
```

Figure 32: Clients of `two_way_sort`.

Routine `client_2` calls `two_way_sort` on a four-element array and checks that its first element is **False** after the call. Modular verification cannot prove this assertion: `two_way_sort` has no postcondition, and hence its effects within `client_2` are undefined. The second verification attempt inlines `two_way_sort` and unrolls the loop four times (since it notices that the call is on a four-element array); this proves that the first array element is **False** after the call (first line in Table 31). In all, `two_way_sort` is not to blame because its implementation works correctly for `client_2`. As summarized in

AutoProof

Verify | 6 Successful | 0 Failed | 0 Errors | Filter:

Class	Feature	Information	Pos...	Time...
CLIENT	two_way_sort	Verification successful.		0.14
CLIENT	swap	Verification successful.		0.13
CLIENT	client_1	Verification successful after inlining. (see original error)		0.69
		You might need to weaken the precondition.		
		Precondition count_positive may be violated on call to two_way_sort.	17	
CLIENT	client_2	Verification successful after inlining. (see original errors)		0.21
		You might need to strenghten the loop invariant or postcondition of called featu...		
		Precondition may be violated on invocation of item.	31	
		Check may be violated (untagged).	31	

Figure 33: AutoProof showing feedback of two-step verification.

the second line of Table 30, the user can either just be happy with the result or endeavor to write down a suitable postcondition for `two_way_sort` so that the correctness proof can be generalized to arrays of arbitrary length.

Suppose we provide a postcondition that specifies sortedness using Eiffel's across syntax:

```
across 1..(a.count−1) as k all
        (a[k] = a[k+1]) or (a[k] ≠ a[k+1] and a[k] = False)
```

Modular verification of `two_way_sort` fails to prove this postcondition because the loop invariant at line 7 does not say anything about the array content. Two-step verification makes a second attempt where it unrolls the loop a finite number of times, say 5, and inlines `swap`. The situation is in the second entry of Table 31: we cannot verify that the arbitrary bound of five iterations generally holds (that is $j - i + 1 \leq 5$ holds before the loop), but the success of unrolling in this limited case suggests that `two_way_sort`'s implementation is correct. If we want to get to a general proof, we should improve the loop invariant, and this is precisely the suggestion that two-step verification brings forward.

5.5.5 Evaluation

The examples of the previous sections have demonstrated the kind of feedback two-step verification provides. This section contains a preliminary evaluation of the scalability of two-step verification, and of its benefits in terms of reduced annotation burden.

Table 34 lists the example programs. The first labeled column after the program name contains the size of the implementation (not counting specification elements) in lines of code. The rest of the table is split in two parts: the first one contains data about two-step verification; the second one the same data about modular verification. The data reported includes: the amount of specification necessary to successfully verify the example (num-

		TWO-STEP						MODULAR					
		P	*Q*	*I*	*A*			*P*	*Q*	*I*	*A*		
	code	*spec*				*Boogie*	*time*	*spec*				*Boogie*	*time*
1. Maximum	32	1	3	0	0	1541	2.06	1	3	4	0	712	1.05
		4						8					
2. Sum and Max	32	3	1	0	0	1619	2.18	3	1	5	0	720	1.04
		4						9					
3. Two-way Sort	44	2	1	0	0	1803	2.35	2	1	5	0	742	1.06
		3						8					
4. Dutch Flag	45	2	1	0	0	1955	2.94	2	1	10	0	786	1.14
		3						13					
5. LCP	30	3	4	0	0	1585	2.10	3	4	7	0	730	1.05
		7						14					
6. Priority queue	119	0	0	0	7	2896	3.35	5	13	5	7	1088	1.56
		7						30					
7. Deque	127	0	0	0	11	1856	2.51	7	18	4	11	1230	1.59
		11						40					
8. Binary Search	48	0	0	0	1	2479	3.21	2	3	0	1	672	0.99
		1						6					
Total	*477*	*11*	*10*	*0*	*19*	*15734*	*20.70*	*25*	*44*	*40*	*19*	*6680*	*9.48*
		40						*128*					

Table 34: Comparison of two-step and modular verification on selected examples.

ber of annotations, split into preconditions P, postconditions Q, invariants I (loop invariants in examples 1–5; class invariants in examples 6–8), and intermediate assertions A); the size (in lines) of the Boogie code generated by AutoProof; and the time in seconds. Two-step verification includes modular verification as first step, but normally requires less specification to be successful; correspondingly, the Boogie code and the time in the first part of the table sum up both steps.

The examples include: (1) finding the maximum in an array, from the COST 2011 verification competition [28]; (2) computing maximum and sum of the elements in an array, from the VSTTE 2010 verification competition [87]; (3) the two-way sort algorithm of Section 5.2, from the VSTTE 2012 verification competition [68]; (4) Dikstra's Dutch national flag algorithm [55]; (5) computing the longest common prefix of two sequences, from the FM 2012 verification competition [80]; (6) a priority queue implementation, from Tinelli's verification course [145]; (7) a double-ended queue [88, Vol. 1, Sec. 2.2.1]; and (8) the binary search algorithm of Section 5.2, from the software verification benchmarks [156].

In the experiments with the algorithms 1–5, two-step verification succeeds with loop unrolling of depth $n = 6$, which corresponds to input arrays of the same length. The outcome suggests either to use the unrolled loop, for inputs of bounded length; or to write a suitable loop invariant. A correct loop

invariant is necessary for modular verification alone to succeed, in which case the proof generalizes to arrays of arbitrary length. We prove the following postconditions: (1) the output is the array maximum; (2) the output sum is less than or equal to the output maximum times the array length; (3) sortedness of the output, as formalized in Section 5.5.4; (4) the output is partitioned in the three flag colors; (5) the output is the longest common prefix of the input array.

In the experiments 6–8 we verify *clients* of the queue, double-ended queue, and binary search, which call some routines and then formalize their expectations on the result with **`assert`**s after the call. The called routines have no specification (in particular, no postcondition); two-step verification verifies the clients using inlining of the callee, and suggests to add postconditions to generalize the proofs. The postconditions are necessary for modular verification alone to succeed.

The evaluation suggests that two-step verification can check the implementation even when little or no specification is given; its feedback may then help write the necessary specifications to generalize proofs for modular verification.

The runtime overhead of performing two verification steps instead of one is roughly linear in all examples; in fact, unrolling and inlining blow up mainly in the presence of recursion. To better assess how they scale, we have repeated two-step verification of examples 3 (using unrolling) and 6 (using inlining in the presence of recursion) for increasing value of the bound n. Table 35 shows the results in terms of size of the generated Boogie code (in lines) and time necessary to verify it (in seconds). Unrolling scales gracefully until about $n = 10$; afterward, the time taken by Boogie to verify increases very quickly, even if the size of the Boogie code does not blow up. Inlining is more sensitive to the bound, since the size of the inlined code grows exponentially due to the conditional branch in `binary_search`'s body; the time is acceptable until about $n = 7$. Notice that the heuristics for the choice of n discussed in Section 5.5.3 would generate running times in the order of tens of seconds, thus enforcing a reasonable responsiveness.

5.5.6 Related Work

The steadily growing interest for techniques and tools that make verification more approachable indicates how some of the most glaring hurdles to the progress of formal methods lie in their applicability. Tools such as Dafny [94], Spec# [20], VCC [38], ESC/Java2 [40, 83], and Why3 [27] define the state of the art in static program verification. Their approaches rely on accurate specifications, which are not easy to write and get right.

	UNROLLING		INLINING	
inlining/unrolling depth n	*Boogie*	*time*	*Boogie*	*time*
3	864	1.07	1201	1.44
4	937	1.13	1822	2.26
5	1010	1.21	3054	4.23
6	1083	1.32	5518	10.93
7	1156	1.52	10446	30.64
8	1229	2.03	20302	112.32
10	1375	4.26	–	–
13	1594	37.52	–	–
15	1667	253.30	–	–

Table 35: Scalability of unrolling and inlining on examples 3 and 6 from Table 34.

One way to ameliorate this situation is inferring specifications automatically using static [33, 89, 71] or dynamic [59, 153, 152] techniques. Specifications dynamically inferred are based on a finite number of executions, and hence may be unsound; this makes them unsuitable for use in the context of static verification. Static techniques can infer sound specifications from the program text; these are useful to document existing implementations, to discover auxiliary assertions (such as loop invariants), or for comparison with specifications independently written, but proving an implementation correct against a specification inferred from it is mostly a vacuous exercise.

The simple implicit contracts that we use in our approach express well-formedness properties of the input program, which are tacitly assumed by programmers reasoning informally about it; therefore, there is no risk of circularity. Some static verifiers use mathematical integers or assume purity of specification functions to have well-formedness by construction; a risk is that, when they are applied to real programming languages, the corresponding semantic gap may leave some errors go unnoticed. ESC/Java2, for example, does not check for overflows [86], nor if specification expressions are executable (for example, null-dereferencing could happen when evaluating a precondition). The Dafny verifier checks well-formedness of pre- and postconditions, and may consequently require users to add explicit contracts to satisfy well-formedness. Our implicit contracts are instead added and checked automatically, without requiring users to explicitly write them. In this sense, they are similar to approaches such as VCC, which models the semantics of the C programming language as precisely as possible.

Besides inferring specifications, another approach to facilitate formal veri-

fication is combining complementary verification techniques. CEGAR model-checking [23], for example, uses model-checking exhaustive verification techniques on approximate program models, combined with a form of symbolic execution to determine whether the failed verification attempts are indicative of real implementation errors or only a figment of an imprecise abstraction. Tools such as DSD-Crasher [50] and our EVE [147] integrate testing and static checking to find when the errors reported by the latter are spurious. Collaborative verification [34] is also based on the combination of testing and static verification, and on the explicit formalization of the restrictions of each tool used in the combination. Two-step verification also integrates the results of different techniques, with the main purpose of improving error reporting and reducing the number of annotations needed, rather than complementing the limitations of specific techniques.

The Spec# system includes a verification debugger [77] to inspect error models when verification fails; more recently, an interpreter for Boogie programs [132] can help find the sources of failed verification attempts. Debuggers for verification can be quite useful in practice, but achieve a lesser degree of automation than two-step verification, since users need to manually inspect and understand the error models using the debugger.

5.6 Polymorphic Calls

The redefinition of a routine r in a descendant class can *strengthen* r's original postcondition by adding an **ensure then** clause, which conjoins the postcondition in the precursor. The example in Figure 36 illustrates a difficulty occurring when reasoning about postcondition strengthening in the presence of polymorphic types. The deferred (abstract) class EXP models nonnegative integer expressions and provides a routine eval to evaluate the value of an expression object; even if eval does not have an implementation in EXP, its postcondition specifies that the evaluation always yields a nonnegative value stored in attribute last_value, which is set as side effect. Classes CONST and PLUS respectively specialize EXP to represent integer (nonnegative) constants and addition. Class ROOT is a client of the other classes, and its main routine attaches an object of subclass CONST to a reference with static type EXP, thus exploiting polymorphism. Similar issues occur when a descendant class *weakens* a some routine r's precondition with an **require else** clause.

The verification goal is proving that, after the invocation *e.eval* (in class ROOT), eval's postcondition in class CONST holds, which subsumes the **check** statement in the caller. Reasoning about the invocation only based on the static type EXP of the target e guarantees the postcondition last_value ≥ 0,

```
deferred class EXP
feature
  last_value: INTEGER
  eval
    deferred
    ensure
      last_value ≥ 0
    end
end
```

```
class PLUS inherit EXP feature
  left, right: EXP
  eval do
      left.eval; right.eval
      last_value := left.last_value +
              right.last_value
    ensure then
      last_value = left.last_value +
            right.last_value
    end
invariant
  left ≠ right ≠ Current
end
```

```
class CONST inherit EXP
feature
  value: INTEGER
  eval do
      last_value := value
    ensure then
      last_value = value
    end
invariant
  value ≥ 0
end
```

```
class ROOT
feature
  main
    local
        e: EXP
    do
      e := create {CONST}.make (5);
      e.eval
      check e.last_value = 5 end
    end
end
```

Figure 36: Nonnegative integer expressions.

which is however too weak to establish that `last_value` is exactly 5.

Other approaches, such as Müller's [112], have targeted these issues in the context of Hoare logics, but they usually are unsupported by automatic program verifiers. In particular, with the Boogie translation of polymorphic assignment implemented in Spec#, the assertion **check** `e.last_value` = 5 **end** in class `ROOT` can be verified only if `eval` is declared *pure*; `eval` is, however, not pure. The Spec# methodology selects the pre and postconditions according to static types for non-pure routines: the call `e.eval` only establishes `e.last_value` ≥ 0, not the stronger `e.last_value` = 5 that follows from `e`'s dynamic type `CONST`, unless an explicit cast redefines the type `CONST`. The rest of the section describes the solution implemented in AutoProof, which handles contracts of redefined routines.

5.6.1 *Polymorphism in Boogie*

The Boogie translation implemented in AutoProof can handle polymorphism appropriately even for non-pure routines; it is based on a methodology for agents [120] and on a methodology for pure routines [52, 99]. The rest of the section discusses how to translate postconditions and preconditions of redefined routines in a way that accommodates polymorphism, while still supporting modular reasoning.

Postconditions

The translation of the postcondition of a routine r of class X with result type T (if any) relies on an auxiliary function `post.X.r`:

```
function post.X.r (h1, h2: HeapType; c: ref; res: T) returns (bool);
```

which predicates over two heaps (the pre and post-states in r's postcondition), a reference `c` to the current object, and the result `res`. r's postcondition in Boogie references the function `post.X.r`, and includes the translation $\nabla_{post}(X.r)$ of r's postcondition clause syntactically declared in class `X`:[4]

```
procedure X.r (Current: ref) returns (Result: T);
  free ensures post.X.r (Heap, old(Heap), Current, Result);
  ensures ∇_post(X.r);
```

`post.X.r` is a **`free ensures`**, hence it is ignored when proving r's implementation and is only necessary to reason about usages of r.

The function `post.X.r` holds only for the type X; for each class Y which is a descendant of X (and for X itself), an axiom links r's postcondition in X to r's strengthened postcondition in Y:

```
axiom (∀ h1, h2: HeapType; c: ref; r: T •
    type_of(c) <: Y ⟹(post.X.r(h1, h2, c, r) ⟹ ∇_post(Y.r)));
```

The function `type_of` returns the type of a given reference; hence the postcondition predicate $post.X.r$ implies an actual postcondition $\nabla_{post}(Y.r)$ according to c's dynamic type.

In addition, for each redefinition of r in a descendant class Z, the translation defines a fresh Boogie procedure $Z.r$ with corresponding postcondition predicate $post.Z.r$ and axioms for all of Z's descendants.

Preconditions

Eiffel also supports *weakening of preconditions*. Therefore, the precondition of a routine can also depend on the dynamic type. We use a similar translation as for the postcondition. Given a routine r of type X, a precondition

[4] The translation differs for calls to **`Precursor`** (**super** in Java and **base** in C#).

predicate is generated and used in the signature of the generated Boogie procedure:

```
function pre.X.r(h: HeapType; c: ref) returns (bool);
```

which predicates over one heap and a reference `c` to the current object. `r`'s precondition in Boogie references the function `pre.X.r`, and it includes the translation $\nabla_{pre}(X.r)$ of r's precondition originally declared in class X:

```
procedure X.r (Current: ref) returns (Result: T);
  requires pre.X.r(Heap, Current);
  free requires ∇pre(X.r)
```

Conversely to the postcondition, establishing r's precondition is a responsibility of callers of r; clients have to establish the precondition determined by the dynamic type—captured by the function `pre.X.r`—, whereas the precondition originally given in X is given as a **free requires** and is only used to prove r's implementation.

```
axiom (∀ h: HeapType; c: ref •
    type_of(c) <: Y ⟹(∇pre(Y.r) ⟹pre.X.r(h, c)));
```

To establish `pre.X.r`, it is enough to establish any of the clauses $\nabla_{pre}(Y.r)$.

5.6.2 An Example of Polymorphism with Postconditions

Figure 37 shows the essential parts of the Boogie translation of the example in Figure 36. The translation of routine `eval` in lines 3–6 references the function `post.EXP.eval`; the axioms in lines 8–17 link that function to r's postcondition in `EXP` (lines 8–10) and to the additional postcondition introduced in `CONST` (lines 11–13) and `PLUS` (lines 14–17) for the same routine.

The rest of the figure shows the translation of the client class `ROOT`. To verify the assertion on line 31 the verifier will use the fact that the reference `e` is of type `CONST` (line 26), that the postcondition function `post.EXP.eval` holds thanks to the postcondition of `eval` (line 4), and the axiom generated for the specific subtype (line 11). With this information, the verifier can establish that the assertion holds.

5.6.3 Related Work

Spec# [20] uses dynamic postconditions for pure functions, taking the dynamic object type into account. In contrast, our approach works for functions that have side-effects and for adaptations of preconditions as well.

Rapid Type Analysis [14] can be used to determine all possible types of an object at a call site. The algorithm analyses the full system and is therefore

```
function post.EXP.eval(h1, h2: HeapType; c: ref) returns (bool);

procedure EXP.eval(current: ref);
  free ensures post.EXP.eval(Heap, old(Heap), current);
  ensures Heap[current, last_value] >= 0;
  // other specification omitted

axiom (∀ h1, h2: HeapType; o: ref •
  type_of(o) <: EXP ⟹
  (post.EXP.eval(h1, h2, o) ⟹ (h1[o, last_value] >= 0)) );
axiom (∀ h1, h2: HeapType; o: ref •
  type_of(o) <: CONST ⟹
  (post.EXP.eval(h1, h2, o) ⟹ h1[o, last_value] =h1[o, value]) );
axiom (∀ h1, h2: HeapType; o: ref •
  type_of(o) <: PLUS ⟹
  (post.EXP.eval(h1, h2, o) ⟹ (h1[o, last_value] =
    h1[h1[o, left], last_value] + h1[h1[o, right], last_value])) );

implementation ROOT.main (Current: ref) {
    var e: ref;
  entry:
    // translation of: create {CONST} e.make (5)
    havoc e;
    assume Heap[e, allocated] =false;
    Heap[e, allocated] :=true;
    assume type_of(e) =CONST;
    call CONST.make(e, 5);
    // translation of: e.eval
    call EXP.eval(e);
    // translation of: check e.last_value = 5 end
    assert Heap[e, last_value] =5;
    return;
}
```

Figure 37: Simplified Boogie translation of the Eiffel classes in Figure 36.

non-modular. For local variables that are only available in the context of a single routine this analysis would be suitable even for modular verification.

Using abstract interpretation [45], Rapid Atomic Type Analysis [104] can be used to infer precise types of numeric variables, even for dynamically typed languages.

5.7 Exceptions

Eiffel's exception handling mechanism is different from most other object-oriented programming languages such as C# and Java.[5] This section presents Eiffel's exception mechanism (Section 5.7.1), discusses how to annotate exceptions (Section 5.7.2), and describes the translation of Eiffel's exceptions to Boogie (Section 5.7.3) with the help of an example and a small case study (Section 5.7.4).

The methodology described here applies to the Eiffel exception mechanism as described in the Eiffel ECMA standard [56]. Since the Eiffel compilers existing today are still using the old exception mechanism, this methodology is not implemented in AutoProof.

5.7.1 How Eiffel Exceptions Work

Eiffel exception handlers are specific to each routine, where they occupy an optional **`rescue`** clause, which follows the routine body (**`do`**). A routine's **`rescue`** clause is ignored whenever the routine body executes normally. If, instead, executing the routine body triggers an exception, control is transferred to the **`rescue`** clause for exception handling. The exception handler will try to restore the object state to a condition where the routine can execute normally. To this end, the body can run more than once, according to the value of an implicit variable **`Retry`**, local to each routine: when the execution of the handler terminates, if **`Retry`** has value **`True`** the routine body is run again, otherwise **`Retry`** is **`False`** and the pending exception propagates to the **`rescue`** clause of the caller routine[6].

Figure 38 illustrates the Eiffel exception mechanism with an example. The routine `attempt_transmission` tries to transmit a message by calling `unsafe_transmit`; if the latter routine terminates normally, then also

[5] Other work has formalized the semantics of Java exceptions [113] and compared it against Eiffel's [118].

[6] This is how Eiffel's exception mechanism is described in the current Eiffel specification draft by the Eiffel ECMA committee. The main Eiffel compiler is using the older exception mechanism, where **`retry`** is a statement that immediately exits the rescue clause

`attempt_transmission` terminates normally without executing the **`rescue`** clause. On the contrary, an exception triggered by `unsafe_transmit` transfers control to the **`rescue`** clause, which re-executes the body `max_attempts` times; if all the attempts fail to execute successfully, the attribute `failed` is set and the exception propagates.

```
1  attempt_transmission (m: STRING)
2    local
3      failures: INTEGER
4    do
5      failed := False
6      unsafe_transmit (m)
7    rescue
8      failures := failures + 1
9      if failures <max_attempts then
10       Retry := True
11     else
12       failed := True
13     end
14   end
```

Figure 38: An Eiffel routine with exception handler.

5.7.2 Specifying Exceptions

The postcondition of a routine with **`rescue`** clause specifies the program state both after normal termination and when an exception is triggered. The two post-states are in general different, hence we introduce a global Boolean variable **`ExcV`**, which is **`True`** if and only if the routine has triggered an exception. Using this auxiliary variable, specifying postconditions of routines with exception handlers is straightforward. For example, the postcondition of routine `attempt_transmission` in Figure 38 says that `failed` is **`False`** if and only if the routine executes normally:

```
attempt_transmission (m: STRING)
  ensure
    ExcV implies failed
    not ExcV implies not failed
```

The example also shows that the execution of a **`rescue`** clause behaves as a loop: a routine `r` with exception handler `r` **`do`** s_1 **`rescue`** s_2 **`end`** behaves as the loop that first executes s_1 unconditionally, and then repeats s_2 ; s_1

until s_1 triggers no exceptions or **Retry** is **False** after the execution of s_2 (in the latter case, s_1 is not executed anymore). To reason about such implicit loops, we introduce a *rescue invariant* [121, 122]; the rescue invariant holds after the first execution of s_1 and after each execution of s_2 ; s_1. A suitable rescue invariant of routine `attempt_transmission` is:

```
rescue invariant
  not ExcV implies not failed
  (failures <max_attempts) implies not failed
```

5.7.3 Eiffel Exceptions in Boogie

The auxiliary variable **ExcV** becomes a global variable in Boogie, so that every assertion can reference it. The translation also introduces an additional precondition `ExcV` = **false** for every translated routine, because normal calls cannot occur when exceptions are pending, and adds ExcV to the modifies clause of every procedure. Then, a routine with body s_1 and rescue clause s_2 becomes in Boogie:

```
  T(s1, excLabel)
excLabel:
  while (ExcV)
  invariant T(I_rescue);
  {
    ExcV := false;
    Retry := false;
    T(s2, endLabel)
    if (¬Retry) { ExcV :=true; goto endLabel; }
    T(s1, excLabel)
  }
endLabel:
```

where $\mathcal{T}(s, l)$ denotes the Boogie translation $\mathcal{T}(s)$ of the instruction s, followed by a jump to label l if s triggers an exception:

$$\mathcal{T}(s,l) \;=\; \begin{cases} \mathcal{T}(s',l)\,;\,\mathcal{T}(s'',l) & \text{if } s \text{ is the compound } s'\,;\,s'' \\ \mathcal{T}(s)\,;\,\textbf{if}\ (\texttt{ExcV})\,\{\textbf{goto}\ \texttt{l};\} & \text{otherwise} \end{cases}$$

Therefore, when the body s_1 triggers an exception, `ExcV` is set and the execution enters the rescue loop. On the other hand, an exception that occurs in the body of s_2 jumps out of the loop and to the end of the routine.

The exception handling semantics is only superficially similar to having control-flow breaking instructions such as *break* and *continue*—available in languages other than Eiffel—inside standard loops: the program locations

where the control flow diverts in case of exception are implicit, hence the translation has to supply a conditional jump after every instruction that might trigger an exception. This complicates the semantics of the source code, and correspondingly the verification of Boogie code translating routines with exception handling. This makes it not only harder for the verifier to prove a routine, but also for the programmer to understand the exact semantics of the code.

5.7.4 Example and Case Study

Figure 39 shows the translation of the example in Figure 38. To simplify the presentation, Figure 39 renders the attributes `max_attempts`, `failed`, and `transmitted` (set by `unsafe_transmit`) as variables rather than locations in a heap map. The loop in lines 23–37 maps the loop induced by the **`rescue`** clause, and its invariant (lines 24 and 25) is the rescue invariant.

We have done a small case study with a set of routines presented in Meyer's book [107] when describing Eiffel exceptions and a second set of classes that are part of the EiffelStudio compiler. To verify them, we extended the original contracts with postconditions to express the behavior when exceptions are triggered, and with rescue invariants. The translation of the examples to Boogie was done by hand, as the described methodology is not implemented in our verifier. The results of the case study are summarized in the following table:

Example name	Classes	LOCEiffel	LOCBoogie	Time [s]
Textbook OOSC2	1	106	481	2.33
Runtime ISE	4	203	561	2.32

The most difficult part of verifying these example was inventing rescue invariants. Even when the examples are simple, the rescue invariants may be subtle, because not only is the loop to reason about implicit, but they also have to include clauses both for normal and for exceptional termination.

5.7.5 Related Work

Other object-oriented languages such as Java or C# have exception mechanisms that allow to catch exceptions for arbitrary code blocks. If exceptions are not catched, they are thrown to the calling context. This execution model—in absence of unusual situations like using jump instructions in the catch block—does not introduce implicit loops and therefore does not need exception invariants.

To deal with exceptional executions of routines, Java-based verifiers such as Krakatoa [105] and OpenJML [39] allow postconditions for the case of

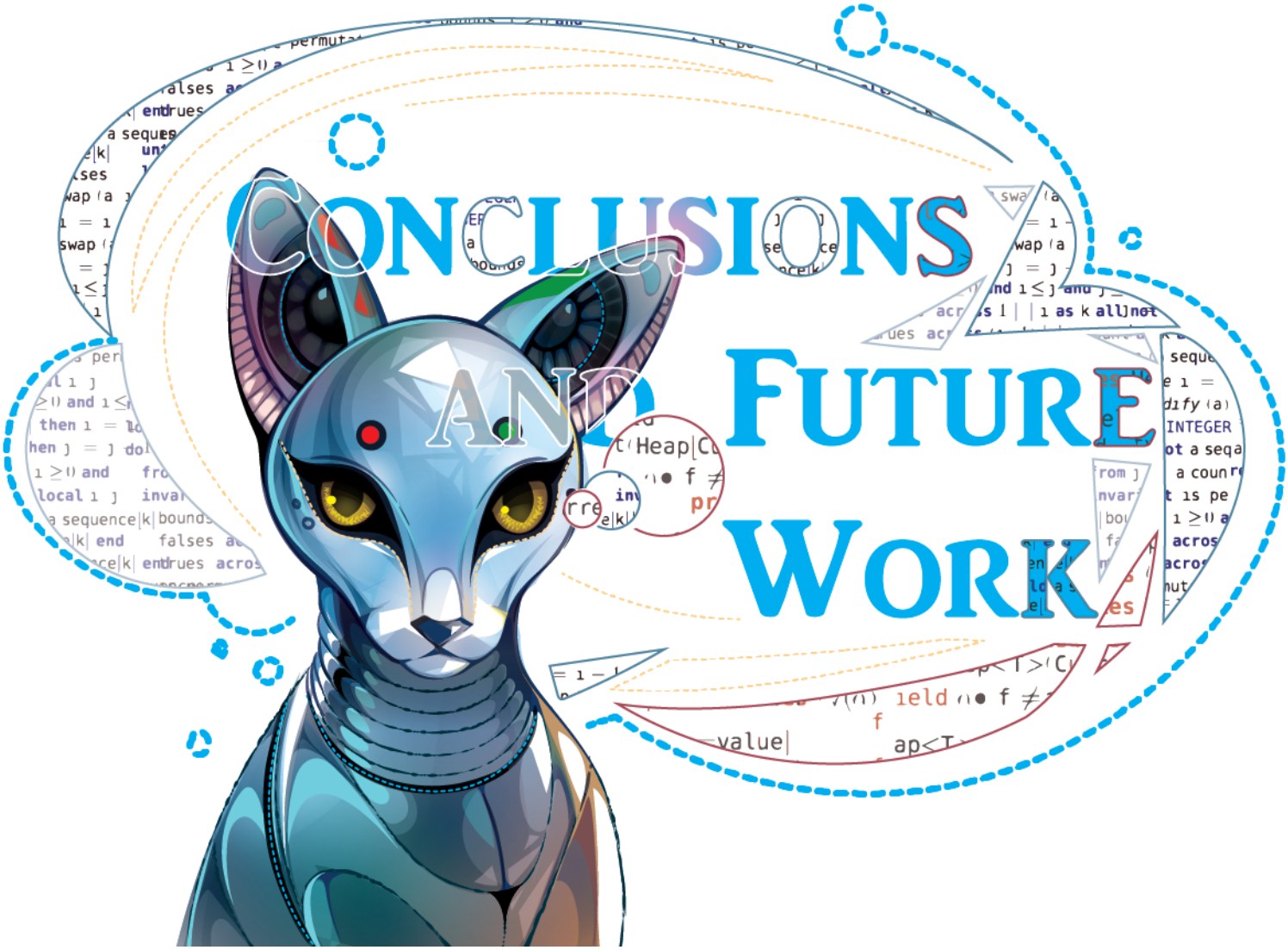
```
1  var max_attempts: int;
2  var failed: bool;
3  var transmitted: bool;
4
5  procedure unsafe_transmit (m: ref);
6    free requires ExcV =false;
7    modifies ExcV, transmitted;
8    ensures ExcV ⟺¬transmitted;
9
10 procedure attempt_transmission (m: ref);
11   free requires ExcV =false;
12   modifies ExcV, transmitted, max_attempts, failed;
13   ensures ExcV ⟺failed;
14
15 implementation attempt_transmission (m: ref)
16 {
17     var failures: int;
18     var Retry: bool;
19   entry:
20     failures :=0; Retry := false; failed :=false;
21     call unsafe_transmit (m); if (ExcV) { goto excL; }
22   excL:
23       while (ExcV)
24       invariant ¬ExcV ⟹¬failed;
25       invariant (failures < max_attempts) ⟹¬failed;
26       {
27         ExcV := false; Retry :=false;
28         failures :=failures + 1;
29         if (failures < max_attempts) {
30           Retry :=true;
31         } else {
32           failed :=true;
33         }
34         if (¬ Retry) {ExcV :=true; goto endL;}
35         failed :=false
36         call unsafe_transmit (m); if (ExcV) { goto excL; }
37       }
38   endL:
39     return;
40 }
```

Figure 39: Boogie translation of the Eiffel routine in Figure 38.

an uncaught exception. This is equivalent to using an exception value to conditionally trigger some postconditions in the regular case and other postconditions in the exceptional case.

Spec# provides a special annotation to signal to clients of a routine which exceptions will be triggered if the caller does not fulfill particular precondition clauses [100]

Chapter 6



6.1 Conclusions

This thesis presented work in the area of automated software verification with a focus on usability and tool support.

To combine verification tools in an IDE we have developed a scoring system to aggregate results in a single interface. The system works by having

each tool assign a score for each routine and provide a confidence in the soundness of its own result. A single routine score computed based on the results and confidence of each tool. To create a class score, routine scores are combined based on a routine's importance. The developer can investigate the correctness of the system at different granularity levels: per class, per feature, and per tool result. A prototype of this system—the Verification Assistant—has been implemented as part of EVE combining the output of three diverse tools: AutoProof, AutoTest, and Eiffel Inspector. The implementation shows that integration on this abstraction level facilitates the addition of new tools in the system.

With AutoProof we have developed a state-of-the-art auto-active verifier for a real complex object-oriented language. AutoProof supports a powerful methodology for framing and class invariants which make it applicable in practice to idiomatic object-oriented patterns. In addition, AutoProof supports a methodology to deal with function objects (agents) and polymorphism. Integrating two-step verification, implicit contracts, inlining, and unrolling, AutoProof tries to cater not just to the verification experts but also be accessible for novices. We have evaluated AutoProof on a rich collection of benchmark problems, all of which are available in an online repository, with a specific focus on verification challenges and object-oriented patterns. AutoProof has also been used to verify full functional correctness of a general-purpose container library. The results attest AutoProof's competitiveness among tools in its league on cutting-edge functional verification of object-oriented programs. We have given an in-depth description of several solutions to verification challenges, highlighting various aspects of using AutoProof in practice. It is possible to use AutoProof for these challenges, although inside knowledge of the back-end verifier is necessary sometimes for a successful verification.

The work on AutoProof has gone on over several years. The prototype version was created as a proof-of-concept and used an inflexible design that was difficult to extend. To ease further development, a new version of AutoProof was built with usability and extensibility in mind. The implementation uses clear abstractions and exhibits a solid object-oriented design; AutoProof offers extension points to integrate new translation from Eiffel to Boogie, as well as facilities to translate Eiffel directly to custom Boogie code. This allows to build Eiffel libraries with custom Boogie axiomatizations without the need of modifying AutoProof. We provide various interfaces for using AutoProof through a graphical user interface, command-line interface, and online and embedded in websites.

We have developed two-step verification, a technique for improving the feedback of automated verifiers. Two-step verification does a second verifi-

cation attempt for each failed verification whilst inlining routine calls and unrolling loops. By combining the outcome of both verification attempts the verifier can improve the error reporting in cases where the implementation of the verified routine is correct and the specification has to be adapted. The evaluation of two-step verification has shown that for small to moderate examples the overhead is manageable.

To support the Eiffel programming language more fully, we have proposed methodologies for handling polymorphic calls and the exception mechanism of Eiffel. Our methodology for polymorphism takes advantage of redefined contracts whenever the dynamic type of a dynamically bound reference can be inferred from the context. This is achieved by using uninterpreted functions representing pre- and postconditions that are linked to the concrete specifications of a routine based on the dynamic type. We have presented a translation of Eiffel's peculiar exception mechanism to Boogie. Routines that trigger exceptions need an additional postcondition to specify the routine's effect in case of an exception and, since the exception mechanism introduces an implicit loop, a rescue invariant is necessary to reason about the exceptional case.

6.2 Future Work

Integrating suggestion tools

Our tool integration in EVE has been limited to verification and analysis tools. Another category of tools that developers benefit from are *contract inference tools* and *fixing tools* such as Daikon [60] or AutoFix [128, 127]. Although these tools cannot be integrated directly with the scoring system implemented in the Verification Assistant, the information collected from the verification tools can be used to target inference and fixing tools to the areas of a program that most benefit from their work. The static verification tools are indicative of the level of specifications of a program element. If static verification fails while testing succeeds, contract inference could be used to improve the available specifications to help the static verifiers. When testing fails as well, fixing tools can try to remove the found bugs. Also, verification tools can be used to validate suggestions. Specifications and fixes generated by suggestion tools can be validated by the verification tools.

The workflow of the tool interactions could therefore be as follows:

- A program element is verified statically. If the verification is successful, additional verifications are only run if the tool's confidence is not 100%

(which indicates unsoundness). If the verification fails, other verification tools are run; in addition, contract inference tools are used on the program elements involved to possibly improve the existing specification.

- A program element is verified dynamically. If the verification is successful, we run additional verification tools only if the score is not yet high enough. If the verification fails a bug was uncovered; fixing tools are used to come up with an automatic fix for the bug.

Language design for verification

```
1  remove
2    require -- regular precondition
3      not is_empty
4    require ("AutoProof") -- precondition only for AutoProof
5      modify_model (Current.sequence)
6    do ···
7    ensure -- regular postcondition
8      count = old count − 1
9    ensure ("ghost") -- non-executable specification
10     sequence = old sequence.but_first
11   end
```

Figure 40: Adding additional semantics to specifications.

Our work on supporting verification of a full object-oriented programming language like Eiffel has taught us that specification based on regular executable code is not sufficient. For the verification of complex properties the language needs support for additional annotations such as framing or termination, but also ghost code with an expressiveness rivaling or even surpassing regular code seems indispensable [65]. A flexible language extension that supports the design and implementation of advanced verification tools natively would be desirable. One approach in this direction would be to split specifications into multiple categories, as indicated in Figure 40. The different types of specifications could add additional semantics to the specifications:

- Separate specifications that are executable or non-executable.
- Highlight specifications that carry addition language semantics such as waiting-conditions in SCOOP programs.

- Mark specifications that are entirely tool-specific.

An additional challenge in this work is to guarantee consistency of specifications of different categories.

Verification of concurrent programs

AutoProof only targets sequential programs. The ever growing importance of concurrent programming makes it worthwhile to investigate extending AutoProof to concurrent programs. Eiffel has adopted the SCOOP concurrency model [107, 116], which is therefore a natural target for AutoProof. Since SCOOP has clearly specified locking based on **separate** arguments and synchronization points based on feature calls on separate entities, execution of individual routines can be handled using the sequential model of Eiffel.

```
1 store (a_buffer: separate BUFFER [INTEGER]; an_element: INTEGER)
2   require
3     not a_buffer.is_full
4   do
5     a_buffer.put (an_element)
6   ensure
7     not a_buffer.is_empty
8     a_buffer.count = old a_buffer.count + 1
9   end
```

Figure 41: Extract from producer-consumer pattern in SCOOP.

Consider as an example the code in Figure 41, an extract from the producer-consumer pattern written in SCOOP. SCOOP guarantees that the separate argument is locked during the execution of the routine, the execution of this isolated routine is therefore equivalent to a sequential execution.

Interesting situations arise in collaborative structures, where one does not always hold a lock on another object, but requires certain properties to hold. The greatest challenge then is to combine the SCOOP model with semantic collaboration [134] ensuring that the guarantees provided by the framing model are retained.

Supporting more domain theories

Given AutoProof's goal of targeting a real programming language, there are few domain-specific features of the Eiffel language that are not fully supported but are used in practice in a variety of programs: reasoning in AutoProof about strings and floating-point numbers is limited by the imprecision

of the verification models of such features. Although strings can be modeled in AutoProof based on the underlying data structure, this approach requires a larger effort of the back-end verifier. A more efficient is possible by bypassing this indirection and use AutoProof's translation capabilities to directly map some language elements to Boogie.

```
1 set_name (n: STRING)
2   require
3     two_letters_minimum: n.sequence.count ≥ 2
4   do
5     name := n
6   ensure
7     name.sequence = n.sequence
8   end
```

Figure 42: Specifications involving strings.

Currently, programs that use strings require developers to write specifications involving the string's model query as shown in Figure 42. This is not just a burden on the developer, also the verifier uses an indirection by modeling the string as an object containing an array of characters. By adding a direct translation for strings in AutoProof, Eiffel strings could be mapped directly to a Boogie array, removing the need to write specifications over model queries while simultaneously making the verification easier for the verifier.

Debugging of failed verifications

When a program fails, developers have debuggers available that allow to inspect the state of the program at the point and the time of the fault. The feedback of a failed verification with AutoProof is currently a single message indicating which assertion could not be verified. To remedy this situation, AutoProof could integrate with the Boogie verification debugger [76]. For the integration, the generated Boogie code needs to be instrumented with special commands. Boogie then capture the state of the internal model at these locations. An additional translation step is necessary to map the Boogie model to variables in the Eiffel program, so the AutoProof output needs to be augmented with additional trace annotations for each generated Boogie variable. A graphical user interface can then be developed that visualizes the verifiers' model and gives users a detailed view of the state of the values the verifier has instantiated at each program location.

Appendix A

AutoProof is an auto-active[1] verifier for the Eiffel programming language; it proves functional correctness of Eiffel programs annotated with contracts. The goal of this tutorial is to show how to verify Eiffel programs with AutoProof through hands-on exercises.

[1] AutoProof tries to achieve an intermediate degree of automation in the continuum that goes from *auto*matic to inter*active*.

Preparations

To use AutoProof locally you can install EVE on your machine. Although it is possible to use the online version of AutoProof to do the verification exercises, some options are not available on the web.

You can download the EVE delivery at

`http://se.inf.ethz.ch/research/eve/builds`

Downloads for the examples and exercises as well as links to using the web interface of AutoProof are available here:

`http://se.inf.ethz.ch/research/autoproof/tutorial`

Structure

Each of the following sections describes in detail the use of AutoProof based on increasingly complex examples. Each example is used throughout one section to explain some of the concepts behind AutoProof and how they are used to verify programs. Each section also has hands-on exercises with verification tasks for one or more programs.

A.1 Verification of Basic Properties

To prove functional correctness automatically, a program needs a machine-readable specification. We are using Eiffel—an object-oriented programming language— which allows one to write contracts as part of the program. Each Eiffel routine is equipped with pre- and postconditions and each class has a class invariant.

We will use the *Account* example to show the basic concepts of AutoProof. The example consists of the two classes `ACCOUNT` and `ACCOUNT_TEST`. The first class models a bank account and the second class consists of two test cases that show proper and improper usage of the class. The full code of class `ACCOUNT` is shown in Figure 43.

First you can look through the example and verify the two classes; all routines, except for the deliberately failing test case, will be successfully verified.

```
note
  description: "Account class."
  model: balance, credit_limit

class ACCOUNT
create make

feature {NONE} -- Initialization

  make
      -- Initialize empty account.
    note
      status: creator
    do
      balance := 0
      credit_limit := 0
    ensure
      balance_set: balance = 0
      limit_set: credit_limit = 0
    end

feature -- Access

  balance: INTEGER
      -- Balance of this account.

  credit_limit: INTEGER
      -- Credit limit of this account.

  available_amount: INTEGER
      -- Amount available on this account.
    note status: functional
    do
      Result := balance + credit_limit
    end

feature -- Element change

  set_credit_limit (limit: INTEGER)
      -- Set 'credit_limit' to 'limit'.
    require
      valid: limit ≥ (0).max(−balance)
      modify_model ("credit_limit", Current)
    do
      credit_limit := limit
    ensure
      limit_set: credit_limit = limit
    end

  deposit (amount: INTEGER)
      -- Deposit 'amount' in this account.
    require
      amount_non_negative: amount ≥ 0
      modify_model ("balance", Current)
    do
      balance := balance + amount
    ensure
      balance_set: balance = old balance + amount
    end

  withdraw (amount: INTEGER)
      -- Withdraw 'amount' from this account.
    require
      amount_not_negative: amount ≥ 0
      amount_available: amount ≤ available_amount
      modify_field (["balance", "closed"], Current)
    do
      balance := balance − amount
    ensure
      balance_set: balance = old balance − amount
    end

feature -- Basic operations

  transfer (amount: INTEGER; other: ACCOUNT)
      -- Transfer 'amount' from this to 'other'.
    require
      amount_not_negative: amount ≥ 0
      amount_available: amount ≤ available_amount
      no_aliasing: other ≠ Current
      modify (Current, other)
    do
      balance := balance − amount
      other.deposit (amount)
    ensure
      balance = old balance − amount
      other.balance = old other.balance + amount
      credit_limit = old credit_limit
      other.credit_limit = old other.credit_limit
    end

invariant
  limit_not_negative: credit_limit ≥ 0
  balance_not_credit: balance ≥ −credit_limit

end
```

Figure 43: Account example.

A.1.1 Input Language

Eiffel Programs and Contracts

Here we give a short overview of the Eiffel programming language based on the *Account* example.

Class definition Eiffel is object-oriented. Classes are defined with the **class** keyword. If no inheritance clause is given (as in this example), then the class implicitly inherits from the class ANY, which serves as a common ancestor for all classes.

```
class
  ACCOUNT
```

Constructors To define constructors for the class, you can use the **create** keyword, followed by a comma-separated list of constructors called *creation routines*. If no constructor is defined, the routine default_create will implicitly become the only creation routine. In our example the routine make will be the creation procedure.

```
create
  make
```

Features and visibility Routines and attributes (together called *features*) are defined in feature blocks using the **feature** keyword. Feature blocks can declare a visibility restriction by indicating a list of class names in curly braces. For example the first feature block restricts the access of the make routine to NONE, essentially hiding the routine from all other classes (no class can inherit from NONE). The other feature blocks do not have any access restriction and thus the features inside these feature blocks are public. It is common to name feature clauses by adding a comment using the double-dash -- comment style (there are no multi-line comments in Eiffel).

```
feature {NONE} -- Initialization
feature -- Access
feature -- Basic operations
```

Attributes Attributes are defined with an attribute name followed by the type of the attribute. For all features it is common to add a comment on the line following the feature declaration.

```
balance: INTEGER
  -- Balance of this account.
```

Routines A routine declaration consists of the routine name, optional parameters, optional return type, optional precondition, routine body and optional postcondition. The precondition denoted by the **require** keyword and postcondition denoted by the **ensure** keyword are the specification of the routine. The precondition holds prior to the execution of the routine, and the postcondition holds afterwards. Therefore the precondition is the responsibility of the client of the routine, whereas the postcondition has to be established by the routine itself. If a pre- or postcondition is omitted, the routine will have an implicit pre- or postcondition of **True**.

```
set_credit_limit (limit: INTEGER)
    -- Set 'credit_limit' to 'limit'.
  require
    valid: limit ≥ (0).max(−balance)
    modify_model ("credit_limit", Current)
  do
    credit_limit := limit
  ensure
    limit_set: credit_limit = limit
  end
```

Assertion tags Each assertion, be it a precondition, a postcondition, a class or loop invariant, or an intermediate check instruction, can have an *assertion tag*. These tags are useful for debugging, as the feedback from AutoProof will specify the tag of violated assertions.

Class invariants Class invariants are written at the end of a class using the **invariant** keyword. Class invariants define the state of a consistent object and hold by default whenever an object is visible to other classes, for example at the beginning and end of each public routine.

```
invariant
  credit_limit_not_negative: credit_limit ≥ 0
  balance_not_below_credit: balance ≥ −credit_limit
```

There are more details on how to write an Eiffel program and what specification can be written for the verification with AutoProof; this will be explained throughout the rest of the tutorial.

AutoProof Annotations

AutoProof supports tow forms of custom annotations: note clauses for features and classes, and dummy routines made available through `ANY`.

Note clauses are used to denote special types of routines and attributes that influence the verification like creation routines (see Section A.1.5) or ghost features (see Section A.2.6). Additionally, note clauses are used to disable defaults for implicit pre-/postconditions of the ownership methodology (see Section A.3.4).

The second form of AutoProof annotations are dummy features (routines and functions with empty implementation) that can be used in assertions or regular code. These features are defined in class `ANY` and are available everywhere. AutoProof gives special semantics to these features, for example to specify modifies clauses (see Section A.1.4).

You can look at the AutoProof manual for a complete listing of custom annotations of both note clauses and dummy features[2].

A.1.2 Basic Properties

Booleans

The Eiffel boolean operations **`not`**, **`and`**, **`or`**, **`xor`**, and **`implies`** are supported by AutoProof. The semi-strict operators **`and then`** and **`or else`** are also supported with the correct semantics that the right-hand side only needs to be valid if the left-hand side does not already define the overall value of the expression.

Integers

The Eiffel integer operations $+$, $-$, $*$, // (quotient of integer division), and \\ (remainder of integer division) are supported by AutoProof. Integers in AutoProof can be modeled in two modes, either as mathematical integers or as machine integers. By default integers will be modeled as mathematical integers, though AutoProof can also check overflows of bounded integers (see Section A.2.8).

The Eiffel comparison operations on integers $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$ are all supported.

[2] `http://se.inf.ethz.ch/research/autoproof/manual/#annotations`

References

Comparison of objects always uses reference equality. The standard equality operator `a = b` and inequality operator `a ≠ b` work as expected; object equality `a ~ b` and inequality `a /~ b` are not supported and will fall back to reference equality when used.

A.1.3 Models

AutoProof supports *model-based contracts*. Models are used to express the *abstract state space* of a class and describe its changes. To define the model of a class you add a `model` annotation to the **`note`** clause of the class. The model may only consist of attributes of the class.

```
note
  model: balance, credit_limit
class ACCOUNT ···
```

This makes the two attributes `balance` and `credit_limit` model fields of the class.

The idea behind model-based contracts is to have an abstract and concise yet expressive way to specify the interface of a class. When using models you use the class invariant to describe object validity in terms of the model attributes. The effect of each procedure is expressed by relating the pre-state of the model fields to their post-state. In addition you can express the framing specification in terms of the model fields.

The Mathematical Model Library (MML, see Section A.2.1) can be used to model complex behavior. Also, ghost attributes might be introduced to define abstract behavior in terms of other functions or attributes and can then be used as model fields (see Section A.2.6).

A.1.4 Framing

The framing model that AutoProof uses is based on *modifies clauses*. The `ACCOUNT` class deliberately used three different ways of specifying the modifies clause to demonstrate the differences between them.

modify_model **(fields, objects)**

Using *`modify_model`* you can specify that model fields may change during the execution of a routine. You can specify one or more model fields by providing as first parameter a manifest string with the name of the model attribute or a manifest tuple with multiple manifest strings. The second

parameter is either a single object, a single set of objects of type MML_SET, or a manifest tuple with mixed objects or sets of objects.

```
deposit (amount: INTEGER)
  require
    ...
    modify_model ("balance", Current)
  do ··· end
```

This routine is allowed to modify the model field balance of the Current object.

The effect of *modify_model* is as follows: each model attribute specified in the *modify_model* clause *as well as each non-model attribute* can be modified in the routine. All model fields that are not listed remain unchanged. This means in turn that for clients all non-model attributes are potentially modified even though they are not listed in the modifies clause.

modify_field (fields, objects)

With *modify_field* you specify directly which attributes may be changed by a routine. As before, you can specify one or more attribute names by providing as first parameter a manifest string with the name of the model attribute or a manifest tuple with multiple manifest strings. The second parameter is again either a single object, a single set of objects of type MML_SET, or a manifest tuple with mixed objects or sets of objects.

```
withdraw (amount: INTEGER)
  require
    ...
    modify_field (["balance", "closed"], Current)
  do ··· end
```

This routine is allowed to modify the attributes balance and closed of the Current object.

This way of specifying the modifies clause is lower-level than specifying which model fields may change. This is also the reason we are required to add the ghost field closed in the example shown here. The closed field is a boolean flag that is True whenever an object is in a consistent state (see Section A.3 for details).

modify (objects)

The third option to specify modifies clauses is to give a list of objects which can be modified without limiting the modifications to certain attributes or model fields. For this modifies clause you can specify mixed objects or sets of objects.

```
transfer (amount: INTEGER; other: ACCOUNT)
  require
    ...
    modify (Current, other)
  do ··· end
```

This routine is allowed to modify all attributes of **Current** and other.

Since the objects may be modified freely, you have to specify the full effect on the modified objects. For example the `transfer` procedure of the account example, the postcondition not only describes the effect on the `balance` attribute of the two objects but also has clauses to specify that the `credit_limit` attribute does not change. This is for demonstration purposes only, it would be a better design to use *`modify_model`* instead (*try to change it!*).

Giving an empty tuple as argument—*`modify`* ([])—denotes that nothing may be modified, i.e., that the routine is *pure*.

Default Modifies Clauses

When no modifies clause is given a default modifies clause is used based on the type of routine:

- For **procedures** (routines without a return value), the default modifies clause is *`modify`* (**Current**). So all attributes can be modified in a procedure if no specific modifies clause is given.

- For **functions** (routines with a return value), the default modifies clause is *`modify`* ([]). Therefore, by default, all functions are *pure*.

When you overwrite the default modifies clause for procedures, for example to modify an object passed as parameter, and you want to be able to modify the **Current** object as well, you will need to add *`modify`* (**Current**) to the modifies clause (or a more specific version when only a subset of the attributes needs to be modifiable).

Combining modifies annotations

You can add several modifies annotations to a modifies clause. The set of modifiable objects and attributes is the union of all modifies annotations.

A.1.5 Routine Annotations

Creation Procedures

Creation procedures can be used as regular routines as well. Therefore, AutoProof will verify all creation routines twice, once as creation routines and once as regular routines. The context of the verification is different for the two verifications, as for example for creation routines all attributes are initialized to their default values before the routine is executed.

You can instruct AutoProof to verify a creation routine only once by adding a `creator` annotation. This denotes the routine as being creation-only and AutoProof will not verify it as a regular routine.

```
make
  note
    status: creator
  do ··· end
```

Marks `make` to be only a creation routine.

Functional Functions

AutoProof supports a special type of function, consisting of only a single assignment to **`Result`**. To declare such a function you have to add a *`functional`* annotation to the function. These functions are defined by their implementation and have an implicit postcondition; given an implementation **`Result`** := `x` the implicit postcondition will be **`Result`** = `x`.

```
available_amount: INTEGER
  note
    status: functional
  do
    Result := balance + credit_limit
  end
```

Marks `available_amount` to be *functional*, therefore only consisting of a single assignment to **`Result`**.

A.1.6 Debugging Verification

The only feedback given by AutoProof is whether a routine is successfully verified or if some specific assertions could not be proven. When the verification fails it can be necessary to find out which facts the verifier could establish or even guide the verifier to the right conclusion. For this you can

use intermediate assertions (**check** instructions in Eiffel). During the debugging process it can also be beneficial to *assume* specific facts and thus limit the possible executions that the verifier considers during the proof.

Assertions

Using Eiffel's **check** instruction you can add an intermediate assertion that will be verified by AutoProof. This can help to check if you have the same understanding of the state at a program point as the verifier. You can add multiple expressions to a single check instruction, and each expression can be equipped with a tag. AutoProof will show the tags in error messages.

`check tag: expr end`	Check instruction to establish if `expr` holds.

Note that it is possible that when you have multiple consecutive assertions successfully verified, removing an intermediate assertion will make the verification of later assertions fail. In these cases you have to keep the assertion in order to guide the verifier towards the successful verification.

Assumptions

Eiffel does not support assumptions out of the box. To write an assumption in AutoProof, you have to write a check instruction with the special tag `assume`. AutoProof will assume the expression for the rest of the routine without checking it.

You can use assumptions to limit the executions considered by the verifier. For example by assuming **False** in a branch of a conditional instruction the verification of that branch will always succeed.

`if ··· then`
`  ···`
`else`
`  check assume: False end`
`end`
	Ignores all code path that go through the **else** branch.

Another way to use assumptions to limit executions it by restricting the state space of otherwise unrestricted values. This can be used for example to ignore executions where an array is empty.

`check assume: not a.is_empty end`	Ignores executions where `a` is empty.

Inconsistencies

It can happen that verification succeeds due to inconsistent contracts or assumptions. If you for example have a routine with the precondition `a` >0 and an additional class invariant `a` <0 (or an assumption `a` <0 in the body of the routine), your specification is inconsistent. This is essentially equivalent to an assumption of **`False`** and the verifier will be able to derive any fact from it, including false ones.

A quick (though not completely safe) check for inconsistencies is to add an assertion or postcondition **`False`** to your routine. If the verifier manages to prove the assertion, this is a sign for an inconsistency in the specification.

A.1.7 Hands-On: Clock

The `CLOCK` class is modeling a clock counting seconds, minutes and hours of a day. The class contains routines to create the clock, set the time, and increase the time.

Task 1: Add a `model` declaration to define the abstract model.

Task 2: Add a class **`invariant`** to restrict the attribute values.

Task 3: Add a precondition to the creation procedure `make`.
You should be able to verify `make` and `test_make`.

Task 4: Add the specification to the `set_*` procedures.
You should be able to verify the `set_*` and `test_set` procedures.

Task 5: Add the specification to the `increase_*` procedures.
You should be able to verify both classes completely.

A.2 Verification of Algorithmic Problems

An important aspect in the verification of programs is verifying algorithms. In this section we will focus on the verification of algorithmic problems on arrays, such as searching and sorting. The concepts needed to verify array algorithms are also necessary for other types of algorithms.

We use the algorithm of finding the maximum element of an integer array as an example. The code is shown in Figure 44. You can look through the example again and verify it. In the rest of this section we will explain in detail how one verifies such an algorithm.

A.2.1 Mathematical Model Library

To express complex mathematical properties, AutoProof supports the *Mathematical Model Library*. This library consists of classes modeling sets, bags (or multisets), sequences, maps, intervals, and relations. You can find an API description of these classes online[3].

MML classes do not have an implementation and should therefore only ever be used for specifications (using *ghost fields* and *ghost code* as discussed in Section A.2.6). They have an efficient axiomatization in the back-end verifier, and are therefore well suited to be used with AutoProof.

MML Types

The most important MML types are:

- `MML_SET` [`G`]: A set contains distinct objects. Each element can only be contained once and the order is irrelevant.
- `MML_SEQUENCE` [`G`]: A sequence is an ordered list of elements. Indexing starts at 1.
- `MML_BAG` [`G`]: A bag (or multiset) is a set where each element can appear multiple times. The order of elements is irrelevant.

Shorthand Notations

Several shorthand notations exist to declare sets and sequences making the use of MML classes easier.

- Sets of type `MML_SET` [`ANY`] can be declared using the Eiffel manifest tuple notation: `s` := [`a`, `b`].

[3] `http://se.inf.ethz.ch/research/autoproof/reference/`

```
class MAX_IN_ARRAY
feature -- Basic operations

  max_in_array (a: SIMPLE_ARRAY [INTEGER]): INTEGER
      -- Find the maximum element of 'a'.
    require
      array_not_empty: a.count >0
    local
      i: INTEGER
    do
      Result := a[1]
      from
        i := 2
      invariant
        i_in_bounds: 2 ≤ i and i ≤ a.sequence.count + 1
        max_so_far: across 1 |..| (i−1) as c all a.sequence[c.item] ≤ Result end
        in_array: across 1 |..| (i−1) as c some a.sequence[c.item] = Result end
      until
        i >a.count
      loop
        if a[i] >Result then
          Result := a[i]
        end
        i := i + 1
      variant
        a.count − i
      end
    ensure
      is_maximum: across 1 |..| a.count as c all a.sequence[c.item] ≤ Result end
      in_array: across a.sequence.domain as c some a.sequence[c.item] = Result end
    end

end
```

Figure 44: *Maximum in array* example.

- Sets of type `MML_SET [G]` can be declared using the Eiffel manifest array notation: `s := ≪a, b≫`.
- Sequences of type `MML_SEQUENCE [G]` can be declared using the Eiffel manifest array notation: `s := ≪a, b≫`.
- Use `{MML_SET [G]}.empty_set` to declare an empty set.
- Use `{MML_SEQUENCE [G]}.empty_sequence` to declare empty sequences.

For the last two shorthands it is not possible to use the empty array notation ≪≫ due to the intricacies of Eiffel typing.

A.2.2 SIMPLE_ARRAY and SIMPLE_LIST

When you want to use arrays or lists in verification, you need classes that have a fully specified interface. The classes from EiffelBase do not offer this, therefore when verifying algorithms with AutoProof, you should use the two provided classes `SIMPLE_ARRAY [G]` and `SIMPLE_LIST [G]`. Both classes have a ghost model field `sequence` of type `MML_SEQUENCE [G]` and all features are specified in terms of the model. You can find an API description of these classes online[4].

To make it easier for AutoProof to deal with specifications involving these classes you should use the `sequence` model field when writing complex assertions involving the container contents. For example the loop invariant of the `max_in_array` function is written as:

```
2 ≤ i and i ≤ a.sequence.count + 1
across 1 |..| (i−1) as c all a.sequence[c.item] ≤ Result end
across 1 |..| (i−1) as c some a.sequence[c.item] = Result end
```

Were you to replace `a.sequence` with just `a`, AutoProof would not verify the routine anymore (*try it!*).

A.2.3 Quantifiers

Eiffel supports bounded universal and existential quantifiers with the **across** expression. In our example, where we find the maximum in an array, we can use this to express the desired postcondition that all elements in the array are smaller or equal to the result. Universal quantification is done using the **across**..**all** expression. With the Eiffel interval expression 1 |..| `a.count` we can quantify over all integers between (and including) 1 and `a.count`.

```
across 1 |..| a.count as c all a.sequence[c.item] ≤ Result end
```

The across loop uses a cursor, therefore we have to use `c.item` to access the current element of the iteration. For the correctness of the algorithm we also have to express that the result is an element of the array, not just larger than all elements. We can do this with an existential quantification using Eiffel's **across**..**some** loop.

```
across a.sequence.domain as c some a.sequence[c.item] = Result end
```

[4] http://se.inf.ethz.ch/research/autoproof/reference/

In the last example we used the `domain` query for the quantification to show that you can use different approaches to reach the same goals. This query defined in `MML_SEQUENCE` returns a set of integer values that contains all index values of the sequence and is therefore equivalent to using an interval from 1 to `a.count` (all MML sequences are indexed from 1).

AutoProof supports the following domains for quantification:

- Integer intervals. The quantified variable will be of type `INTEGER`.
- Sets of type `MML_SET` [`G`]. The quantified variable will be of type `G`.
- Sequences of type `MML_SEQUENCE` [`G`]. The quantified variable will be of type `G`. This is equivalent to quantifying over the `range` of a sequence.
- Objects of type `SIMPLE_ARRAY` [`G`] or `SIMPLE_LIST` [`G`]. The quantified variable will be of type `G`. This is equivalent to quantifying over the `sequence` of the array or list.

A.2.4 Termination

AutoProof will verify termination of loops and direct recursive calls (indirect recursion is not checked). To prove termination you can define *loop variants* for loops or *decreases clauses* for loops and recursive routines.

Loop Variant

The loop variant is an integer expression that is non-negative and decreases with each loop iteration. This implies that the loop can only be executed a finite number of times.

```
loop ···
variant
  a.count − i
end
```

The loop variant decreases each loop iteration and stays non-negative.

AutoProof infers loop variants of simple loops. For example a loop with exit condition `a >b` will have an inferred loop variant of `b − a`. In the example in Figure 44 specifying the variant is not necessary.

Decreases Clause

In complex algorithms it is possible that an integer value is not enough to express the loop variant. For these cases AutoProof supports ***decreases***

clauses. A decreases clause can contain multiple arguments of type INTEGER, MML_SET, or MML_SEQUENCE. The semantics of a decreases clause is that in each loop iteration the tuple that contains all the elements of the decreases clause needs to become lexicographically smaller while remaining bounded from below. The lower bound is 0 for integers and is the empty set or empty sequence for sets and sequences.

The decreases clause for a loop is written in the loop invariant.

```
from ···
invariant
  decreases (a.count − i)
until ···
```

Decreases clause equivalent to the previous loop variant.

For recursive functions the decreases clause is added to the precondition. Otherwise it behaves like the decreases clause for loops: at each recursive call the value of the decreases clause must become smaller while remaining bounded.

```
f (a: SET [INTEGER]; b: INTEGER)
  require
    decreases (a, b)
  do ··· end
```

Decreases clause of a recursive function.

Non-termination

Sometimes it is not desirable to prove termination of an algorithm. For these cases you can add an empty decreases clause to the loop or recursive function and AutoProof will skip the termination check.

```
from ···
invariant
  decreases ([])
until ···
```

A possibly non-terminating loop.

A.2.5 Hands-On: Linear and Binary Search

With the knowledge we have so far we now verify algorithms searching an element in an array. These algorithms do not change the array and are therefore *pure*, thus simplifying the specification.

Linear Search

Task 1: Add the loop variant to verify that the loop terminates.
You should be able to verify `linear_search` in its current form.

Task 2: Add postconditions to `linear_search` to verify the test class.
You should be able to verify the test class.

Task 3: Add loop invariants to verify the postcondition.
You should be able to verify both classes completely.

Binary Search

Task 1: Add loop invariants to verify that all array accesses are valid.

Task 2: Add the loop variant to verify that the loop terminates.
You should be able to verify `binary_search` in its current form.

Task 3: Add precondition to require input arrays to be sorted.

Task 4: Add postconditions to `binary_search` to verify the test class.
You should be able to verify the test class.

Task 5: Add loop invariants to verify the postcondition.
You should be able to verify both classes completely.

Recursive Binary Search

Task 1: Add the specification to `binary_search` (you can reuse the specification of the iterative version).
You should be able to verify the test class.

Task 2: Add precondition to `binary_search_recursive_step` to require the input array to be sorted and to verify that all array accesses are valid.

Task 3: Add a decreases clause to to prove termination of the recursion.
You should be able to verify `binary_search_recursive_step` in its current form.

Task 4: Add postconditions to `binary_search_recursive_step` to verify the algorithm.
You should be able to verify both classes completely.
Note: you might need intermediate assertions to verify the postcondition.

A.2.6 Ghost State and Ghost Functions

The next examples—iterative and recursive binary search—have preconditions that the input array is sorted. Writing an expression that expresses this property directly in the precondition can become unwieldy. It is beneficial to write helper functions that capture such properties with a meaningful name and that allow reuse of the function.

The *sorted* property was expressed over the *sequence* of the array, which is of type `MML_SEQUENCE`. As mentioned before (see Section A.2.1), MML types are not executable and can only be used for specification purposes. We call code that is used only for specification purposes *ghost code*; ghost code is never executed and only interpreted by the verifier.

To write expressive specifications AutoProof supports ghost code in the form of *ghost functions*, using *ghost attributes*, and writing *lemmas*. Ghost code should never influence executable code, therefore assignments from ghost code to regular attributes is not allowed. AutoProof does **not** enforce this currently, so using ghost code outside specifications may lead to undefined behavior.

Ghost Functions

Ghost functions are useful to write helper functions usable in specifications, for example to express that a sequence is sorted. To mark a function as *ghost* you add a **note** clause with `status`: *ghost*. If the function is also *functional*, the note clause can be shortened by combining the two `status` properties.

```
is_sorted (s: MML_SEQUENCE [INTEGER]): BOOLEAN
    -- Is 's' sorted?
  note
    status: functional, ghost
  do
    Result := across 1 |..| s.count as i all
                across 1 |..| s.count as j all
                  i.item ≤ j.item implies s[i.item] ≤ s[j.item] end end
  end
```

With ghost function like the one shown above we can simplify contracts and promote reuse of specification constructs, for example in iterative and recursive binary search (*try it!*).

Ghost State

Ghost state is introduced by having *ghost attributes*. These attributes can be used like regular attributes in contracts, frame conditions, code, and as model

fields. Most commonly you would use ghost attributes to define model fields that are then related to existing attributes or other objects through class invariants and other contracts. A linked list could for example represents its contents in form of a sequence using a ghost attribute to store the sequence and then declaring this attribute to be a model field.

To declare a ghost attribute you need to add a **note** clause to the attribute. The Eiffel syntax for doing this is the following:

```
sequence: MML_SEQUENCE [INTEGER]
  note status: ghost
  attribute
  end
```

Lemmas

Intermediate assertions are not always sufficient for difficult proofs. In these cases you can use *lemma* procedures to support verification. Calling a lemma procedure has the same effect as calling other procedures: the verifier asserts the precondition and assumes the postcondition. Lemmas can therefore be used to add $A(x) \implies B(x)$ to the fact space, where $A(x)$ is the precondition and $B(x)$ is the postcondition of the lemma.

Lemmas are implicitly *ghost* and *pure*. You declare a lemma using a special **note** clause.

```
lemma (x)
  note status: lemma
  require
    A(x)
  do
    -- Proof that A(x) implies B(x)
  ensure
    B(x)
  end
```

Lemmas are proven like a regular procedures. You might need to *implement* a proof; sometimes you can use recursion in a lemma which is akin to an induction proof.

A.2.7 Accessing Pre-state

Eiffel allows the use of **old** expressions in postcondition to express the effect of a routine in relation to the pre-state. This syntax is limited, as it cannot be used in **across** expressions or in the body of the routine (e.g. in loop invariants).

AutoProof offers an extension to the **old** mechanism through a ghost query `old_` defined in `ANY`. This query can be used anywhere in the code or in the postcondition to reference the value of an expression in the pre-state of the routine.

check `s.old_[i] = s[i]` **end**	Assertion that item `s [i]` is unchanged.

A.2.8 Integer Overflows

AutoProof can check a program for integer overflows. By default overflow checking is disabled. You can enable it among AutoProof's options or, if you use the command line version, with the `−overflow` command line option.

A.2.9 Hands-On: Sorting

The next examples are about sorting of arrays. The algorithms shown here are in-place algorithms that operate on the array directly. As a preliminary exercise we look at the notion of permutation of arrays and how to express this in AutoProof.

Permutation

Task 1: Find the correct encoding of permutation (only one is correct).

Task 2: For each incorrect encoding try to find two sequences that successfully pass the check instruction while not being real permutations.

Gnome Sort

Task 1: Add the frame specification, pre- and postcondition to `gnome_sort`.
Add implementation of `is_part_sorted`.
You should be able to verify the test class.

Task 2: Add loop invariants to verify that all array accesses are valid.

Task 3: Add loop invariants to verify the postcondition.
You should be able to verify both classes completely.

Task 4: Enable overflow checking and verity absence of overflows.

Insertion Sort

Task 1: Add a precondition and loop invariant to `insertion_sort` and the precondition to `swap` to verify that all array accesses are valid.

Task 2: Add the loop variants to verify that the loops terminate.
You should be able to verify `insertion_sort` and `swap` in its current form.

Task 3: Add the postcondition to `insertion_sort`.
You should be able to verify the test class.
You might want to introduce helper functions.

Task 4: Add the postcondition to `swap` and the necessary loop invariants to verify the postcondition.
You should be able to verify both classes completely.

Task 5: Enable overflow checking and verity absence of overflows.

A.3 Object Consistency and Ownership

For this section we use an unbalanced binary tree as an example. Each tree node has a value and one or two children. The `maximum` function returns the maximum element in the tree. An excerpt of the tree node class is shown in Figure 45.

A.3.1 State of an Object

AutoProof uses an invariant model where objects can be in a consistent state or in a potentially inconsistent state. Consistent objects are *closed* and their class invariants are guaranteed to hold. Inconsistent objects are *open* and their class invariant is potentially violated. Objects can only be modified when they are open, changing the value of an attribute is not allowed when an object is closed.

AutoProof supports a dynamic ownership model where objects can be *owned* by other objects. The ownership relations can evolve during runtime. Objects that are unowned or whose owner is open are called *free*. We define a shorthand for objects that are *closed* and *free* calling them *wrapped*. When an object is in a consistent state, the ownership tree rooted in that object is guaranteed to be consistent as well.

To model the object consistency and ownership relation, each objects has a boolean ghost field `closed`, a ghost field `owner` pointing to the potential

```
1   class TREE_NODE
2
3   create
4     make, make_with_children
5
6   feature {NONE} -- Initialization
7
8     make (a_value: INTEGER)
9         -- Initialize node.
10      note
11        status: creator
12      do
13        value := a_value
14      ensure
15        value_set: value = a_value
16        no_left: left = Void
17        no_right: right = Void
18      end
19
20    make_with_children (a_value: INTEGER;
21            a_l, a_r: TREE_NODE)
22        -- Initialize node.
23      note
24        status: creator
25        explicit: contracts
26      require
27        a_l.is_wrapped
28        a_r.is_wrapped
29
30        modify (Current)
31        modify_field ("owner",[a_l,a_r])
32      do
33        value := a_value
34        left := a_l
35        right := a_r
36      ensure
37        value_set: value = a_value
38        left_set: left = a_l
39        right_set: right = a_r
40        default_is_closed: is_wrapped
41      end
42
43  feature -- Access
44
45    value: INTEGER
46        -- Value of this node.
47
48    left, right: TREE_NODE
49        -- Left and right node (Void if none).
50
51  feature -- Basic operations
52
53    maximum: INTEGER
54        -- Maximum value of this tree.
55      require
56        decreases (sequence)
57      do
58        Result := value
59        if left ≠ Void then
60          check owns.has (left) end
61          Result := Result.max (left.maximum)
62        end
63        if right ≠ Void then
64          check owns.has (right) end
65          Result := Result.max (right.maximum)
66        end
67      ensure
68        max: across sequence.domain as i all
69                    sequence[i.item] ≤ Result end
70        exists: sequence.has (Result)
71      end
72
73  feature -- Model
74
75    sequence: MML_SEQUENCE [INTEGER]
76        -- Sequence of values.
77      note
78        status: ghost
79      attribute
80      end
81
82  invariant
83    owns_def: owns = {like owns}[[left,right]] / Void
84    sequence_def: sequence =
85      (if left = Void
86         then {like sequence}.empty_sequence
87         else left.sequence end) +
88      {like sequence}[≪value≫] +
89      (if right = Void
90         then {like sequence}.empty_sequence
91         else right.sequence end)
92  end
```

Figure 45: Excerpt of the binary tree example.

owner of the object, and a ghost field `owns` that contains the set of all owned objects. The relationship between the `owns` set and `owner` field is guaranteed to hold for objects in a consistent state. A special case is **`Void`** which is always `open`. **`Void`** can therefore never be owned and must not be part of the `owns` set.

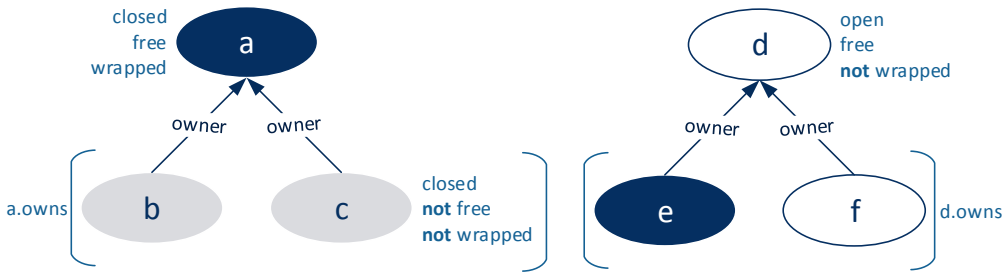


Figure 46: Example of an object structure at run-time.

We illustrate the possible object states on the example object structure of Figure 46. The object structure consist of six objects:

- Object `a` is *closed*, therefore its class invariant is guaranteed to hold. It does not have an owner and thus is *free*. As it is both free and closed, it is also *wrapped*. The object `a` owns the two objects `b` and `c`. This is defined through its `owns` set, i.e. `a.owns` = {`b`, `c`}.

- Objects `b` and `c` are both owned by `a`, so their `owner` ghost field points to `a`. Since they are *owned* they are *not free*. As their owner `a` is *closed*, the two objects `b` and `c` are *closed* as well, as they are in `a`'s ownership domain.

- Object `d` is *open* and may potentially be in an inconsistent state, so its class invariant is not guaranteed to hold. It does not have an owner and is therefore *free*. The object `d` owns the two objects `e` and `f`, defined through its `owns` set.

- Object `e` is *closed* and therefore consistent. Its owner is `d`, but since `d` is *open*, `e` is considered to be *free*. Being both *closed* and *free* means that `e` is *wrapped*.

- Object `f` is *open* and potentially inconsistent. Analogous to `e` it is *free*, as its owner `d` is *open*.

The example illustrates the difference between ownership trees of *open* and *closed* objects. While ownership trees of consistent objects are guaranteed to be consistent as well—all objects in the ownership tree including the root object are *closed*—, this property does not hold for ownership trees of potentially inconsistent objects. The ownership tree of an *open* object may contain objects that are *open* and objects that are *closed*.

A.3.2 Object State Queries

AutoProof offers ghost functions that can be used to query an object's state in assertions and specifications:

- `is_wrapped`: `BOOLEAN` – is the object wrapped (closed and free)?
- `is_free`: `BOOLEAN` – is the object free (unowned or owner is open)?
- `is_open`: `BOOLEAN` – is the object open (potentially inconsistent)?
- `closed`: `BOOLEAN` – is the object closed (consistent)?
- `owner`: `ANY` – owner of the object.
- `owns`: `MML_SET` [`ANY`] – set of owned objects.
- `inv`: `BOOLEAN` – does full invariant of the object hold?
- `inv_only` (`t`): `BOOLEAN` – does the invariant with tag *t* hold?
- `inv_without` (`t`): `BOOLEAN` – does the invariant except for tag *t* hold?

The last two functions `inv_only` and `inv_without` allow to reuse specification constructs. The argument to these functions is a list of manifest strings containing invariant tags. Given the class of Figure 45 the condition that the `sequence` is consistent can be written as `inv_only` ("sequence_def"). This helps reduce the annotation burden for classes with comples class invariants.

A.3.3 Encoding Ownership

Ownership in AutoProof is used by adding objects to and removing objects from the `owns` set. The usual way of doing this is by defining the `owns` set as part of the class invariant. The binary tree example has the following class invariant:

```
owns_def: owns = {like owns}[[left, right]] / Void
```

The owns set consists of the two objects `left` and `right` unless they are **`Void`**. As in other situations, the encoding of the `owns` set influences the ability of AutoProof to reason about it. In the `maximum` function of the binary tree we have introduced two assertions `owns.has (left)` and `owns.has (right)` in the respective branches. Where we to remove these check instructions AutoProof would fail in verifying the function (*try it!*).

This is due to the use of the set removal operation / **`Void`**, which makes the reasoning about the set more difficult and forces us to help the verifier in the proof. Were we to use a different encoding of the `owns` set in the class invariant we could remove these assertions. The following encoding is more verbose but better suited for the verifier:

```
owns =
  if left = Void then
    if right = Void then {like owns}[[]] else {like owns}[[right]] end
  else
    if right = Void then {like owns}[[left]] else {like owns}[[left, right]] end
  end
```

With this encoding there is no need for the intermediate check instructions anymore (*try it!*).

A.3.4 Wrapping and Unwrapping

The two ghost procedures `wrap` and `unwrap` are used to change an object from being *unwrapped* to being *wrapped* and vice versa. Figure 47 gives an overview of how the object consistency changes when these procedures are called.

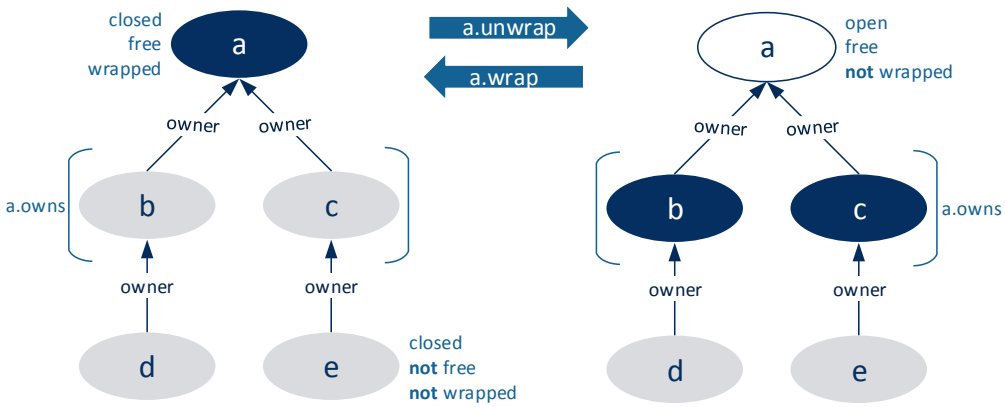


Figure 47: Change of object state on wrapping and unwrapping.

Since wrapping and unwrapping changes the boolean ghost field `closed`,

that field must be writable when either of these procedures are called. This is also the reason we had to add the field `closed` to the modifies clause in the `withdraw` procedure of the *account* example (see Section A.1.4).

`wrap` and `unwrap` are axiomatized in the background theory of the verifier. Their definition in the class `ANY` does therefore not reflect their real semantics. The actual specification for `wrap` and `unwrap` could be written as follows:

```
1  wrap
2      -- Wrap 'Current'.
3    require
4      is_open
5      inv
6      across owns as o all o.item.is_wrapped end
7      modify_field ("closed", Current)
8      modify_field ("owner", owns)
9    ensure
10     is_wrapped
11     across owns as o all o.item.owner = Current end
12   end
```

```
13 unwrap
14     -- Unwrap 'Current.
15   require
16     is_wrapped
17     modify_field ("closed", Current)
18   ensure
19     is_open
20     across owns as o all
21            o.item.is_wrapped end
22   end
```

A.3.5 Defaults

AutoProof uses implicit default contracts, default wrapping, and default assignments of ghost fields to remove the annotation burden. The default contracts and wrapping depend on type and visibility of routines. The default assignment to ghost fields depends on their definition.

Creation routines require by default that all arguments are wrapped. In addition, the **`Current`** object will be implicitly wrapped at the end of the routine body and the postcondition will assert that **`Current`** is wrapped.

```
make (args)
  require
    ∀o ∈ args : o.is_wrapped
  do
    ...
    Current.wrap
  ensure
    Current.is_wrapped
    ∀o ∈ args : o.is_wrapped
  end
```

Pure functions can by default be called on *closed* objects. It is therefore not necessary to unwrap the object before calling a function. Also there is no default postcondition since the objects are not mutated during the execution.

```
pure_function (args): type
  require
```

```
    Current.is_closed
    ∀o ∈ args : o.is_closed
  do ··· end
```

Public procedures and public impure functions require by default that all objects are in a consistent state, as they are callable from client code.

```
public_procedure (args)
  require
    Current.is_wrapped
    ∀o ∈ args : o.is_wrapped
  do
    Current.unwrap
    ...
    Current.wrap
  ensure
    Current.is_wrapped
    ∀o ∈ args : o.is_wrapped
  end
```

Private procedures and private impure functions are only callable from a restricted set of clients, for example from the class itself. Since all public routines by default unwrap the **Current** object, private routines by default assume the object to be open.

```
private_procedure (args)
  require
    Current.is_open
  do ···
  ensure
    Current.is_open
  end
```

Ghost fields that have a definition in the class invariant of the form *fieldname*=*expression* will be assigned implicitly every time the object is wrapped. This is the reason that the two ghost fields `owns` and `sequence` of the binary tree example are never explicitly assigned. AutoProof assigns these fields implicitly at the end of each procedure when the implicit call to `wrap` takes place.

Functional functions are often used in contracts and class invariants. Therefore they may operate on inconsistent objects and have no default preconditions.

Disabling Defaults

It is possible to disable defaults by adding the following `note` clauses to routines and classes:

- `explicit`: `contracts` – disable default pre- and postcondition; applicable to a routine or a class.
- `explicit`: `wrapping` – disable default `unwrap` and `wrap` instructions; applicable to a routine or a class.
- `explicit`: *fieldname* – disable default assignment to field *fieldname* before wrapping; applicable to a class.

A.3.6 Modification of Owned Objects

Modifies clauses define a set of modifiable objects. Whenever an object is modifiable then the *transitive closure* of all owned objects is also modifiable. It is therefore not necessary to add owned objects to a modifies clause when the owner is modifiable. A modifies clause of `modify` (`o`) is sufficient to potentially modify objects owned by `o`. Modifications of owned objects still require that the modified object is unwrapped.

A.3.7 Hands-On: Ring Buffer

The next exercise is about implementing a ring buffer backed by an array. This will highlight how to use ownership and model-based contracts to design a data structure.

Task 1: Add ownership definition for the `data` array to the class invariant.

Task 2: Add class invariants for the bounds of `start` and `free`.

Task 3: Add the model fields, model declaration, and class invariant describing the model.

Task 4: Add the specifications to the routines of `RING_BUFFER`. The tests in the test class can help you to get the specification right.
You should be able to verify both classes completely.

Appendix B

This manual describes how to use AutoProof. AutoProof is an auto-active verifier for the Eiffel programming language that can prove functional correctness of Eiffel programs annotated with contracts. AutoProof is available as part of the Eiffel Verification Environment (EVE) or via an online version.

The manual is also available online:

`http://se.inf.ethz.ch/research/autoproof/manual`.

B.1 AutoProof in EVE

AutoProof is integrated in EVE, a research branch of EiffelStudio.

Installation

Follow the installation instructions of EVE[1].

Open Project

Open and compile an EiffelStudio project using the EVE base library. For this you can download the example project provided[2]. Launch EVE using the *run_eve* script in the EVE delivery and add the example's ecf file using the *add project* button. When you have added the project you can open and compile it.

Open AutoProof tool panel

Click the menu entry *View > Tools > AutoProof*. This will show the AutoProof panel, which can be docked like the other tools in EVE.

Run AutoProof

The Eiffel compilation of the project has to be finished and successful in order to run AutoProof. It is not necessary to do the C compilation when working with AutoProof. When the project is compiled successfully, there are three ways to launch AutoProof:

- You can run AutoProof using the *Verify* button. By default, AutoProof will verify the class or cluster that is shown in the editor pane. You can change this behavior by clicking the down-arrow on the *Verify* button and select either to verify the parent cluster of the item currently shown in the editor, or to verify the whole system (excluding libraries).
- You can pick-and-drop a feature, class, or cluster onto the *Verify* button.
- You can right-click a feature or class, and select *Verify with AutoProof* in the context menu.

Only one execution of AutoProof can run at a time, so when the verification has started the *Verify* button will become inactive.

[1] `http://se.inf.ethz.ch/research/eve`
[2] `http://se.inf.ethz.ch/research/autoproof/manual/template.zip`

Stop AutoProof

When AutoProof is running you can stop it using the red stop button next to the *Verify* button. This can be helpful if you want to cancel a long-running verification.

Filtering Results

There are two ways of filtering the results displayed by AutoProof:

- The three toggle-buttons can be used to show or hide all successful results, failed verifications, or semantic errors.
- The filter box can be used to enter a text. Only results will be shown where this text is contained in either the class name, feature name, or text message. To clear the filter box you can click the red x button next to it.

Options

On the top-right of the AutoProof tool is the options button. Clicking it will display the available options and if the option is enabled or disabled. Clicking any of the options will toggle its value. For an explanation of the options see the AutoProof Options Section B.4.

B.2 AutoProof on the Command-line

The command-line version of AutoProof is available as part of EVE.

Run AutoProof

To run AutoProof via the command-line you have to run the EVE command-line compiler with a valid Eiffel project and add the *-autoproof* option:

`ec.exe` −`config` *ecf-file* −`target` *ecf-target* −`autoproof`

By default all user classes in the system are verified by AutoProof. To select which classes or routines are verified, you can add the class names and routine names as additional command-line arguments. The routine name is composed of the class name together with the routine in the format `CLASS.routine`.

Options

AutoProof has the same options on the command-line as for the graphical version. The available command-line options are listed in the AutoProof Options Section B.4.

B.3 AutoProof on the Web

The online version of AutoProof[3] is integrated in ComCom[4], an online interface to run command-line tools.

Examples

Across the top of the AutoProof interface on Comcom are different examples that can be selected. The examples can be adapted in the browser. To reload the original version, click the reload button on the top-right.

Custom code

The last tab *More AutoProof* can be used to write your own code. In addition to writing your own code you can use command-line options on this tab. For a list of command-line options, see the AutoProof Options Section B.4.

Run AutoProof

To run AutoProof, click the *Run* button below the code area. The results will be shown in the box below the button. Note that changing the example will clear the results box. The verification time is limited to 2 minutes, you will get a result or a time-out message after that time.

Limitations

The ComCom version of AutoProof is limited to examples that consist only of one user-defined class. You can use the classes of the EVE version of EiffelBase in your code, in particular the ones defined in the AutoProof base library[5]. It is not possible to add further libraries.

[3] `http://cloudstudio.ethz.ch/comcom/#AutoProof`
[4] `http://cloudstudio.ethz.ch/comcom/`
[5] `http://se.inf.ethz.ch/research/autoproof/reference`

B.4 AutoProof Options

There are different options which influence the behavior of the AutoProof translation and execution as a whole, as opposed to annotations which only affect individual classes or features (see the Annotations Section B.7 for details). Table 48 lists these options. The command-line switches usually come in pairs: one switch turns the option on and the other turns it off.

B.5 Verification Process

Modularity

AutoProof does routine-level modular verification. Each routine is verified in isolation and only the interface of suppliers are considered during the verification of a routine. A program can only be considered fully verified if all routines are verified individually. By default AutoProof only verifies user-written classes when a program is verified, referenced libraries should be verified separately.

Creation routines

Creation routines in Eiffel double as regular routines and can be called on existing objects. Since routines behave differently depending on whether they are called as a creation routine, e.g. the class invariant is not checked on entry and all attributes are set to their default value for creation routines, these routines are verified twice with AutoProof. The feedback of AutoProof will list these routines twice specifying the context of the verification.

Assumptions

It can be useful to temporarily assume a fact that the verifier should use. This helps in debugging failed verifications, e.g. by directing the verifier in a specific branch. You can write an assumption using a check instruction with the special tag *assume*.

```
check assume: False end
```

Skipping classes or routines

You can skip the verification of single routines or classes by adding a note clause.

Option name	Default	CLI switch	Description
Two-step	Disabled	-twostep -notwostep	Use two-step verification when enabled.
Automatic inlining	Disabled	-autoinline -noautoinline	Inline routines without postcondition automatically when enabled.
Automatic unrolling	Disabled	-autounroll -noautounroll	Unroll loops without loop invariants automatically when enabled.
Postcondition predicates	Disabled	-postpredicate -nopostpredicate	Generate postcondition predicates when enabled.
Overflow	Disabled	-overflow -nooverflow	Check for integer overflows when enabled.
Generate triggers	Disabled	-trigger -notrigger	Generates triggers for across expressions when enabled.
Arithmetic triggers	Disabled	-arithtrigger -noarithtrigger	Uses arithmetic functions for triggers when enabled.
SC defaults	Enabled	-scdefaults -noscdefaults	Use semantic collaboration defaults when enabled.
Bulk verification	Bulk	-bulk -forked	Bulk: all routines are verified at the same time and the results are displayed at the end. Forked: routines are verified in parallel and results are displayed when available.
Timeout		-timeout X	*Command-line only* - Boogie timeout in seconds.
HTML output		-html	*Command-line only* - Produce HTML output.

Table 48: AutoProof options for the graphical and command-line interface

```
note
  status: skip
```

B.6 Language Support

The programming language that AutoProof supports is not exactly the standard Eiffel language. Some features of the Eiffel language are not supported, on the other hand custom annotations have been added to increase the expressiveness of the specification language. All custom annotations are valid syntax in standard Eiffel.

AutoProof supports a large portion of the Eiffel language. When AutoProof encounters code constructs it does not support, it will try to degrade in a correct but incomplete way in order to verify the surrounding program. In any case, a special message will be displayed. Here is a comprehensive list of supported and unsupported instructions, expressions, and language concepts. The color coding represents the level of support for the construct: green represents full support, yellow partial support, and red no support.

B.6.1 Instructions

Name	Example	Comment
Assignment	`a := b`	
Call	`a.f (x)`	The verification will check if `a` is not Void and if the precondition of `f` holds. Adding tag names to the preconditions of `f` improves the error reporting.

Check	`check`
	`  tag: c`
	`end`
		The verification will check if the condition `c` holds. Adding a tag name improves the error reporting.

Conditional	`if c then`
	...
	`else`
	...
	`end`	

Creation	`create a.f(x)`	The verification will check if the precondition of `f` holds. Adding tag names to the preconditions of `f` improves the error reporting.

Name	Example	Comment
Debug	`debug` ··· `end`	

Inspect	`inspect a`
	`when x then` ···
	`when y..z then` ···
	`else` ···
	`end`	

From loop	`from` ···
	`invariant`
	  `tag1: i1`
	  `tag2: i2`
	`until c`
	`loop` ···
	`variant v end`
		The verification will check if the invariants `i1` and `i2` hold after the (possibly empty) `from` block is executed and after every loop iteration. If a loop variant `v` or *`decreases`* annotation is provided then it is checked that the variant is non-negative and is reduced in every loop iteration. Adding tag names to the loop invariants improves the error reporting.

Across loop	`across a as c`
	`loop` ···
	`end`
		Across loops can be expressed with from loops.

Retry	`retry`	The retry statement is only used in exception handling, which is not supported.

B.6.2 Expressions

Name	Example	Comment
Access	`a`	

Across	`across set as c`
	  `all` ··· `end`
	`across` 1 |..| 5 `as c`
	  `some` ··· `end`
		Only specific container types are supported: MML types, `INTEGER_INTERVAL`, `SIMPLE_ARRAY`, and `SIMPLE_LIST`.

Address	$	The address operator is used to interoperate with external C code, which is not supported.
Access	`a`	

Name	Example	Comment
Call	`a.f (x)`	The verification will check if `a` is not Void and if the precondition of `f` holds. Adding tag names to the preconditions of `f` improves the error reporting.
Conditional	`if c then x else y end`	
Creation *(body)*	`create {T}.f (x)`	The verification will check if the precondition of `f` holds. Adding tag names to the preconditions of `f` improves the error reporting.
Creation *(contract)*	`create {T}.f (x)`	The contracts have to be side-effect free, creating objects in the contract is not supported.
Agents	`agent f`	
Manifest array *(body)*	≪a, b≫	This syntax can be used to initialize `SIMPLE_ARRAY`, `MML_SET`, or `MML_SEQUENCE` entities.
Manifest array *(contract)*	≪a, b≫	This syntax can be used to initialize `MML_SET` or `MML_SEQUENCE` entities. Otherwise it is not supported as contracts have to be side-effect free.
Manifest string *(body)*	"abc"	AutoProof provides a special string class `V_STRING` which can be initialized with manifest strings.
Manifest string *(contract)*	"abc"	Not supported as contracts have to be side-effect free.
Manifest tuple *(body)*	[a, b]	This syntax can be used to initialize `TUPLE`s or `MML_SET` entities.
Manifest tuple *(contract)*	[a, b]	This syntax can be used to initialize `MML_SET` entities. Otherwise it is not supported as contracts have to be side-effect free.
Object test	`attached {T} a as x`	This syntax can be used to initialize `MML_SET` entities. Otherwise it is not supported as contracts have to be side-effect free.

Name	Example	Comment
Old	`old a`	The old keyword can only be used in postconditions. For accessing the prestate in the routine body AutoProof provides a special query `old_`.

B.6.3 Library support and built-in types

Name	Example	Comment
Boolean values	`True` `False`	
Boolean operations	`a and b or c` `a implies b xor c`	
Integer values	1 −2	
Integer arithmetic	`a + b − c` `a * b // c`	The handling of integers is based on the built-in capabilities of Boogie. AutoProof also checks arithmetic overflows (if enabled).
Floating point values	1.3 −2.4	Floating point values are mapped to Boogie's `real` type. This is only an approximation of floating-point numbers.
Floating point arithmetic	`a + b − c` `a * b // c`	Floating point values are mapped to Boogie's `real` type. This is only an approximation of floating-point numbers.
Character values	'c'	Characters are mapped to the integer code of the character.
Character operations	`a + b` `c.to_upper`	Only basic arithmetic on characters is supported.
Strings	"abc"	AutoProof provides a special string class `V_STRING` which can be initialized with manifest strings.

Base library	`a: SIMPLE_ARRAY`
	`a[1]`
	`a.put (v, 3)`
		An array and a list class are provided specifically for the use in specification: `SIMPLE_ARRAY` and `SIMPLE_LIST`.

B.7 Annotations

The following tables give a summary of the custom annotations supported by AutoProof. If you use the same tag name for multiple values, you can use either multiple note entries or a comma-separated list for the values. For example the following two declarations are equivalent:

```
note                          note
  status: skip, functional      status: skip
                                status: functional
```

B.7.1 Class Annotations

These annotation can be used in the **note** clause of a class.

Name	Example	Comment
Explicit	`explicit: "all"`	Turn off all semantic collaboration defaults in the class.
Explicit Contracts	`explicit: contracts`	Turn off all semantic collaboration default contracts in the class.

Explicit Sets	`explicit: observers`
	`explicit: subjects`
	`explicit: owns`
		Turn off semantic collaboration default invariants for the mentioned sets.

Explicit Wrapping	`explicit: wrapping`	Turn off all semantic collaboration default unwrapping and wrapping instructions in the class.
Manual inv()	`manual_inv: True`	If **True** then you need to insert manual `inv`, `inv_only`, and `inv_without` assertions in the code.
Model	`model: f, g`	List of model queries.
Status: Skip	`status: skip`	Skip verification of whole class.
Theory	`theory: "file.bpl"`	Include the specified theory file in the generated Boogie code.
Type Mapping	`maps_to: "t"`	Map this class to the specified custom Boogie type.
Type Properties	`type_properties: "f"`	Comma-separated list of Boogie functions that define the type properties for reference-type generic parameters.

Name	Example	Comment
Type Invariant	`where: "f"`	Boogie function that defines the type invariant for this type.

B.7.2 Feature Annotations

These annotation can be used in the **`note`** clause of a feature.

Name	Example	Comment
Explicit	`explicit: "all"`	Turn off all semantic collaboration defaults in the feature.
Explicit Contracts	`explicit: contracts`	Turn off semantic collaboration default contracts in the feature.
Explicit Wrapping	`explicit: wrapping`	Turn off semantic collaboration default unwrapping and wrapping instructions in the feature.
Function Mapping	`maps_to: "f"`	Map this feature to the specified Boogie function.

Guard	`guard: True`
	`guard: False`
	`guard: "f"`
		Set the update guard of this attribute to the specified value. The update guard feature has to be *`functional`*.

Inlining	*`inline`*: **`True`** *`inline`*: 4	Inline calls in this routine. If a number is given, then the calls will be inlined up to the specified depth.
Manual inv()	`manual_inv: True`	If set to true then you need to insert manual `inv`, `inv_only`, and `inv_without` assertions in the code.
Status: Creator	`status: creator`	Verify this routine only in the context of a creation routine.
Status: Dynamic	`status: dynamic`	Verify this routine assuming the **`Current`** type might be a subtype. This removes the need to reverify this routine in subclasses.
Status: Functional	`status: functional`	Mark this routine as functional. Functional features can only contain a single instruction that assigns a value to the **`Result`**.

Name	Example	Comment
Status: Ghost	`status: ghost`	Mark the feature as a ghost feature.
Status: Impure	`status: impure`	Functions are supposed to have an empty modifies clause and are therefore pure. By marking a function as impure, you can add a modifies clause.
Status: Inline	`status: inline`	Mark this feature to be always inlined in the caller.
Status: Lemma	`status: lemma`	Mark this feature as a lemma. Lemma features have no automatic unwrapping and wrapping of the Current object and are implicitly ghost.
Status: Skip	`status: skip`	Skip verification of this feature.

B.7.3 Precondition and Loop Invariant Functions

These annotations are used in the precondition of features or the invariant clause of loops.

Name	Example	Comment

Decreases	`decreases (x)`
	`decreases ([x, y])`
	`decreases ([])`
		Specify the variant of a recursive function or a loop. An empty decreases clause (last line) removes the termination check entirely.

Modifies (object)	`modify (Current)` `modify (set)`	Allow this routine or loop to modify all attributes of the given objects.

Modifies (field)	`modify_field ("x", o)`
	`modify_field ("x", set)`
	`modify_field (["x", "y"], o)`
		Allow this routine or loop to modify the specified attributes of the given objects.

Modifies (model)	`modify_model ("x", o)`
	`modify_model ("x", set)`
	`modify_model (["x", "y"], o)`
		Allow this routine or loop to modify the specified models of the given objects.

Name	Example	Comment
Reads	*reads* (o) *reads_field* ("attr", o) *reads_model* ("attr", o)	Allow this functional routine to read the specified objects, attributes or models.

B.7.4 Commands and Queries

These features are used in the regular body of routines.

Name	Example	Comment
Invariant	inv other.inv	Check whether the class invariant of an object holds.
Partial invariant	inv_only ("t1", "t2") o.inv_without ("t")	Check whether part of the class invariant holds. The check takes a list of tag names given by string constants and either just checks these tags (inv_only) or the whole invariant except for the specified tags (inv_without).
Is Free	is_free other.is_free	Returns whether an object is free. Free objects have no owner.
Is Fresh	is_fresh other.is_fresh	Returns whether an object is fresh. Fresh objects were not allocated in the pre-state.
Is Open	is_open other.is_open	Returns whether an object is open. Open objects might not satisfy their class invariant.
Is Wrapped	is_wrapped other.is_wrapped	Returns whether an object is wrapped. Wrapped objects are closed and free.
Unwrapping	unwrap other.unwrap unwrap_all (set)	Unwrap an object or a set of objects.
Wrapping	wrap other.wrap wrap_all (set)	Wrap an object or a set of objects.

Bibliography

[1] ACSL: ANSI/ISO C Specification Language. `http://frama-c.com/acsl.html`, 2015.

[2] Ada 2012. `http://www.ada2012.org/`, 2015.

[3] Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. A unified approach for static and runtime verification: Framework and applications. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, pages 312–326, 2012.

[4] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Craig interpretation. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, pages 300–316, 2012.

[5] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From under-approximations to over-approximations and back. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 157–172, 2012.

[6] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 672–678, 2012.

[7] Ameer Armaly. The D Programming Language. *Linux J.*, 2007(155), March 2007.

[8] AutoProof Manual. `http://se.inf.ethz.ch/research/autoproof/manual`, 2015.

[9] AutoProof on Comcom. `http://cloudstudio.ethz.ch/comcom/#AutoProof`, 2015.

[10] AutoProof on Eiffel 4 Pubs. `http://cloudstudio.ethz.ch/e4pubs`, 2015.

[11] AutoProof on rise4fun. `http://rise4fun.com/AutoProof`, 2015.

[12] AutoProof Tutorial. `http://se.inf.ethz.ch/research/autoproof/tutorial`, 2015.

[13] AutoProof Verified Code Repository. `http://se.inf.ethz.ch/research/autoproof/repo`, 2015.

[14] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, October 6-10, 1996.*, pages 324–341, 1996.

[15] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 1–3, 2002.

[16] John G. P. Barnes. *High Integrity Software - The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.

[17] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 364–387, 2005.

[18] Michael Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, pages 54–84, 2004.

[19] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[20] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

[21] Clark Barrett and Cesare Tinelli. CVC3. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 298–302, 2007.

[22] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[23] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[24] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[25] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, pages 193–207, 1999.

[26] Joshua Bloch. `http://goo.gl/sWLty`, 2006.

[27] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie*, pages 53–64, 2011.

[28] Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen, and Mattias Ulbrich. The COST IC0701 verification competition 2011. In *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*, pages 3–21, 2011.

[29] Eduardo Brito and Jorge Sousa Pinto. Program verification in SPARK and ACSL: A comparative case study. In *Reliable Software Technologiey - Ada-Europe 2010, 15th Ada-Europe International Conference on Reliable Software Technologies, Valencia, Spain, June 14-18, 2010. Proceedings*, pages 97–110, 2010.

[30] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.

[31] Patrice Chalin. Are practitioners writing contracts? In *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, pages 100–113, 2006.

[32] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and esc/java2. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 342–363, 2005.

[33] Bor-Yuh Evan Chang and K. Rustan M. Leino. Inferring object invariants: Extended abstract. *Electr. Notes Theor. Comput. Sci.*, 131:63–74, 2005.

[34] Maria Christakis, Peter Müller, and Valentin Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 132–146, 2012.

[35] Maria Christakis, Peter Müller, and Valentin Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, pages 336–354, 2015.

[36] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, pages 52–71, 1981.

[37] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[38] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 23–42, 2009.

[39] David R. Cok. The OpenJML toolset. In *NASA Formal Methods*, volume 6617. 2011.

[40] David R. Cok and Joseph Kiniry. Esc/java2: Uniting esc/java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, pages 108–128, 2004.

[41] David R. Cok, Joseph R. Kiniry, and Dermot Cochran. ESC/Java2 implementation notes. Technical report, Kind Software, October 2008.

[42] The Coq Development Team. `https://coq.inria.fr/`, 2015.

[43] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[44] Loïc Correnson and Julien Signoles. Combining analyses for C program verification. In *Formal Methods for Industrial Critical Systems - 17th International Workshop, FMICS 2012, Paris, France, August 27-28, 2012. Proceedings*, pages 108–130. 2012.

[45] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.

[46] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282, 1979.

[47] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *Verification,*

Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings, pages 128–148, 2013.

[48] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 150–168, 2011.

[49] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.

[50] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[51] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - A software analysis perspective. In *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, pages 233–247, 2012.

[52] Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 336–351, 2007.

[53] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[54] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[55] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[56] ECMA International. *Standard ECMA-367 - Eiffel: Analysis, Design and Programming Language*. 2nd edition, June 2006.

[57] EiffelStudio IDE. `http://dev.eiffel.com/`, 2015.

[58] Michael D. Ernst. How tests and proofs impede one another: The need for always-on static and dynamic feedback. In *Tests and Proofs, 4th International Conference, TAP 2010, Málaga, Spain, July 1-2, 2010. Proceedings*, pages 1–2, 2010.

[59] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[60] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[61] H.-Christian Estler, Carlo A. Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. Contracts in practice. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 230–246, 2014.

[62] Eiffel Verification Environment (EVE). `http://se.inf.ethz.ch/research/eve/`, 2015.

[63] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 2103–2110, 2010.

[64] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, pages 10–30, 2010.

[65] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 1–16, 2014.

[66] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/-caduceus platform for deductive program verification. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 173–177, 2007.

[67] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 125–128, 2013.

[68] Jean-Christophe Filliâtre, Andrei Paskevich, and Aaron Stump. The 2nd verified software competition: Experience report. In *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, June 30, 2012*, pages 36–49, 2012.

[69] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 234–245, 2002.

[70] Carlo A. Furia. Rotation of sequences: Algorithms and proofs. `http://arxiv.org/abs/1406.5453`, June 2014.

[71] Carlo A. Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation, Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, pages 277–300, 2010.

[72] Carlo A. Furia, Chris Poskitt, and Julian Tschannen. The AutoProof Verifier: Usability by Non-Experts and on Standard Code. In *Proceedings of 2nd Workshop on Formal Integrated Development Environment, F-IDE 2015, Oslo, Norway, June 22, 2015.*, 2015.

[73] FxCop. `http://msdn.microsoft.com/en-us/library/bb429476.aspx`, 2015.

[74] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[75] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[76] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. The boogie verification debugger (tool paper). In *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*, pages 407–414, 2011.

[77] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. The boogie verification debugger (tool paper). In *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*, pages 407–414, 2011.

[78] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 72–83, 1997.

[79] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[80] Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. VerifyThis verification competition. `http://verifythis2012.cost-ic0701.org`, 2012.

[81] John Hunt. Blackboard architectures, 2002. JayDee Technology Ltd 27.

[82] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, pages 304–311, 2010.

[83] Perry R. James and Patrice Chalin. Faster and more complete extended static checking for the java modeling language. *J. Autom. Reasoning*, 44(1-2):145–174, 2010.

[84] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, pages 268–283, 2006.

[85] Joseph R. Kiniry, Alan E. Morkan, Dermot Cochran, Fintan Fairmichael, Patrice Chalin, Martijn Oostdijk, and Engelbert Hubbers. The KOA remote voting system: A summary of work to date. In *Trustworthy Global Computing, Second Symposium, TGC 2006, Lucca, Italy, November 7-9, 2006, Revised Selected Papers*, pages 244–262, 2006.

[86] Joseph R. Kiniry, Alan E. Morkan, and Barry Denby. Soundness and completeness warnings in ESC/Java2. In *SAVCBS*, pages 19–24. ACM, 2006.

[87] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark A. Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st verified software competition: Experience report. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pages 154–168, 2011.

[88] Donald E. Knuth. *The Art of Computer Programming.* Addison-Wesley, 2011.

[89] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 470–485, 2009.

[90] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.

[91] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accomodates both runtime assertion checking and formal verification. pages 262–284, 2002.

[92] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.

[93] K. Rustan M. Leino. This is boogie 2. Technical report, Microsoft Research, June 2008.

[94] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar,*

Senegal, April 25-May 1, 2010, Revised Selected Papers, pages 348–370, 2010.

[95] K. Rustan M. Leino. Verifying concurrent programs with chalice. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, page 2, 2010.

[96] K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 615–622, 2009.

[97] K. Rustan M. Leino and Michał Moskal. Usable auto-active verification. In *Usable Verification Workshop*. `http://fm.csl.sri.com/UV10/`, 2010.

[98] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, pages 491–516, 2004.

[99] K. Rustan M. Leino and Peter Müller. Verification of equivalent-results methods. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 307–321, 2008.

[100] K. Rustan M. Leino and Wolfram Schulte. Exception safety for c#. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, pages 218–227, 2004.

[101] K. Rustan M. Leino and Valentin Wüstholz. The dafny integrated development environment. In *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.*, pages 3–15, 2014.

[102] Francesco Logozzo. Our experience with the codecontracts static checker - (invited tutorial). In *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, pages 241–242, 2012.

[103] Francesco Logozzo, Michael Barnett, Manuel Fähndrich, Patrick Cousot, and Radhia Cousot. A semantic integrated development environment. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*, pages 15–16, 2012.

[104] Francesco Logozzo and Herman Venter. RATA: rapid atomic type analysis by abstract interpretation - application to javascript optimization. In *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 66–83, 2010.

[105] Claude Marché and Christine Paulin-Mohring. Reasoning about java programs with aliasing and frame conditions. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, pages 179–194, 2005.

[106] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[107] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[108] Bertrand Meyer. Seven principles of software testing. *IEEE Computer*, 41(8):99–101, 2008.

[109] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *IEEE Computer*, 42(9):46–55, 2009.

[110] Bertrand Meyer, Carlo A. Furia, and Sebastian Nanz. Software Verification, ETH Zurich. `http://se.inf.ethz.ch/courses/2014b_fall/sv`, 2015.

[111] F. Lockwood Morris and C. B. Jones. An Early Program Proof by Alan Turing. *IEEE Annals of the History of Computing*, 6(2):139–143, 1984.

[112] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.

[113] Peter Müller and Martin Nordio. Proof-transforming compilation of programs with abrupt termination. In *Proceedings of the 2007 Conference Specification and Verification of Component-Based Systems, SAVCBS 2007, Dubrovnik, Croatia, September 3-4, 2007*, pages 39–46, 2007.

[114] Peter Müller and Joseph N. Ruskiewicz. A modular verification methodology for c# delegates. In *Rigorous Methods for Software Construction and Analysis, Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday*, pages 187–203, 2009.

[115] David A. Naumann. Observational purity and encapsulation. In *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 190–204, 2005.

[116] Piotr Nienaltowski and Bertrand Meyer. *Practical framework for contract-based concurrent object-oriented programming.* PhD thesis, ETH Zurich, Switzerland, 2007.

[117] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[118] Martin Nordio. *Proofs and Proof Transformations for Object-Oriented Programs.* PhD thesis, ETH Zurich, Switzerland, 2009.

[119] Martin Nordio, Cristiano Calcagno, and Carlo A. Furia. Javanni: A verifier for javascript. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 231–234, 2013.

[120] Martin Nordio, Cristiano Calcagno, Bertrand Meyer, Peter Müller, and Julian Tschannen. Reasoning about function objects. In *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, pages 79–96, 2010.

[121] Martin Nordio, Cristiano Calcagno, Peter Müller, and Bertrand Meyer. A sound and complete program logic for eiffel. In *Objects, Components, Models and Patterns, 47th International Conference, TOOLS*

EUROPE 2009, Zurich, Switzerland, June 29-July 3, 2009. Proceedings, pages 195–214, 2009.

[122] Martin Nordio, Peter Müller, and Bertrand Meyer. Proof-transforming compilation of eiffel programs. In *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings*, pages 316–335, 2008.

[123] The OpenJML toolset. `http://openjml.org/`, 2015.

[124] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, pages 748–752, 1992.

[125] David Patterson. For better or worse, benchmarks shape a field: technical perspective. *Commun. ACM*, 55(7):104, 2012.

[126] Yu Pei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Automated program repair in an integrated development environment. In *Companion Proceedings of the 37th International Conference on Software Engineering (ICSE)*. ACM, May 2015.

[127] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Trans. Software Eng.*, 40(5):427–449, 2014.

[128] Yu Pei, Yi Wei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Code-based automated program fixing. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 392–395, 2011.

[129] PMD. `http://pmd.sourceforge.net/`, 2015.

[130] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pages 93–104, 2009.

[131] Nadia Polikarpova, Carlo A. Furia, and Bertrand Meyer. Specifying reusable components. In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, pages 127–141, 2010.

[132] Nadia Polikarpova, Carlo A. Furia, and Scott West. To run what no one has run before: Executing an intermediate verification language. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, pages 251–268, 2013.

[133] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, June 24-26, 2015. Proceedings*, 2015.

[134] Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. Flexible invariants through semantic collaboration. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 514–530, 2014.

[135] Sriram K. Rajamani. Verification, testing and statistics. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pages 33–40, 2009.

[136] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74, 2002.

[137] SAVCBS workshop series. `http://www.eecs.ucf.edu/~leavens/SAVCBS/`, 2010.

[138] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272, 2005.

[139] SPARK Pro. `http://www.adacore.com/sparkpro/`, 2015.

[140] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for flexible object invariants. In *IWACO*, pages 1–9. ACM, 2009.

[141] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 298–315, 2011.

[142] International conference on tests and proofs. Springer LNCS, 2007–2010.

[143] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, pages 134–153, 2008.

[144] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 253–262, 2005.

[145] Cesare Tinelli. Formal methods in software engineering. `http://www.divms.uiowa.edu/~tinelli/181/`, 2011.

[146] Julian Tschannen, Carlo A. Furia, and Martin Nordio. AutoProof meets some verification challenges. *International Journal on Software Tools for Technology Transfer*, pages 1–11, February 2014.

[147] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*, pages 382–398, 2011.

[148] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Program checking with less hassle. In *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, pages 149–169, 2013.

[149] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. Autoproof: Auto-active functional verification of object-oriented programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 566–580, 2015.

[150] Julian Tschannen, Carlo Alberto Furia, Martin Nordio, and Bertrand Meyer. Automatic verification of advanced object-oriented features: The autoproof approach. In *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pages 133–155, 2011.

[151] VisualStudio. `https://www.visualstudio.com/`, 2015.

[152] Andrzej Wasylkowski and Andreas Zeller. Mining temporal specifications from object usage. *Autom. Softw. Eng.*, 18(3-4):263–292, 2011.

[153] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 191–200, 2011.

[154] Yi Wei, Serge Gebhardt, Bertrand Meyer, and Manuel Oriol. Satisfying test preconditions through guided object selection. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 303–312, 2010.

[155] Yi Wei, Bertrand Meyer, and Manuel Oriol. Is branch coverage a good measure of testing effectiveness? In *Empirical Software Engineering and Verification - International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*, pages 194–212, 2010.

[156] Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce M. Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. Incremental benchmarks for software verification tools and techniques. In *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, pages 84–98, 2008.

[157] Stefan Zurfluh. Rule-Based Code Analysis. Master's thesis, ETH Zurich, 2014.