Reasoning about object-oriented programs: from classes to interfaces

Bian, J.

Chapter 8

Conclusion

Throughout the main body of the thesis, we implemented a series of studies on exploring ways to apply formal methods systematically for the verification of complex object-oriented libraries such as the Java Collection Framework. We start with specifying and verifying methods in the `java.util.LinkedList` class, but we encounter challenges with methods that take an interface type as a parameter. To address this, we proposed to use histories as method calls and returns to completely determine the concrete state of any implementation and thus can be seen as a way to reason about the interface. The executable history-based (EHB) approach, designed to facilitate history-based reasoning and creating reusable specifications for Java programs, embeds histories and attributes directly as Java objects. This approach could be seamlessly integrated in the KeY theorem prover and avoids the need to change the KeY system itself. However, the EHB approach still has its limitations, particularly when it comes to reasoning about the heap and properties of user-defined attributes, which can require a lot of work due to alias analysis and dynamic footprints. To mitigate this, we introduce the logical history-based (LHB) approach, which models histories as an external abstract data type with functions. This opened up new possibilities for modeling complex behavior in object-oriented programs. Building on the LHB approach, we have developed a history-based refinement theory for reasoning about hierarchy in object-oriented programs. To systematically conclude the thesis, in this final chapter, we first summarize the contributions we have made to addressing the key challenges of formal verification in object-oriented libraries, as formulated in the introductory chapter. Finally, we provide a list of possible directions for future work.

8.1 Summary of contributions

Ensuring that software libraries operate without errors and function as intended has always been a central concern in the field of computer science. This is especially critical given that these libraries serve as the foundation for countless applications and are used by billions of devices worldwide. Formal verification offers a rigorous, mathematically sound way to confirm the accuracy of the software, grounded on clearly defined behavior criteria expressed in formal logic. While formal verification provides robust assurance of software's correctness, unlike testing, it often demands considerable time and resources to define specifications and develop proofs. This thesis has extended the application of the KeY theorem prover to achieve systematic verification of object-oriented libraries of the popular programming language Java. This work may interest non-specialists, as it shows what features of a specification and verification system we need in order to reason about real-world programs. It is also beneficial for beginning users of KeY and Isabelle/HOL, as we introduce and informally explain several key concepts in Chapter 2. We also provide the artifacts and video materials for each chapter to help in reproducing the proofs underlying the results. These materials also help the expert user and the developer of KeY as a 'benchmark' for specification and (automatic) verification techniques. Below, we give a short summary of the contribution for each chapter.

In Chapter 3, we outline the methodology for analyzing an existing Java program to gain a deeper understanding of its behavior. It emphasizes the importance of precise specifications, using the JML for clarity. To validate the program's behavior against these specifications, the chapter advocates for a formal approach supported by the KeY tool, which uniquely allows for comprehensive reasoning on Java programs. This tutorial emphasizes the critical importance of ensuring program correctness in software libraries, particularly in Java's standard library, due to their widespread use and potential for systemic impact.

In Chapter 4, we explore the reasoning about the correctness of Java interfaces, with a particular application to Java's `Collection` interface. We introduce the concept of a *history* as a sequence of method calls and returns as a general methodology for specifying interfaces and verifying clients and implementations of interfaces. This helps us to develop a novel "proving to interface" methodology.

As a proof-of-concept, using the KeY theorem prover, in Chapter 5, the so-called EHB approach has been applied to the core methods of Java's `Collection` interface. The EHB approach is to embed histories and attributes in the KeY theorem prover by encoding them as Java objects on the heap, thereby avoiding the need to change the KeY system itself. We show our approach is sufficient for reasoning about interfaces from the client's perspective, as well as about classes that implement interfaces. However, the EHB approach uses pure methods that rely on the heap, giving rise to additional proof obligations every time these pure methods are used in JML specifications. Moreover, reasoning about the properties of user-defined functions is complex. For instance, the proofs about *multiset* attribute modeled as a pure method take 72 minutes of work.

We then proposed the LHB approach. The LHB approach encodes histories as built-in ADTs with special proof rules, to avoid modeling histories as Java objects. We discuss integrating ADTs in the KeY theorem prover by a new approach to model data types using Isabelle/HOL as an interactive back-end and representing Isabelle theorems as user-defined taclets in KeY. In Chapter 6, we detail on how we designed our specification of the `Collection` interface, and describe in more detail the steps needed to verify several complex example clients. In this chapter, we have seen an application of our technique to the case of history-based reasoning. The main contribution of this chapter is to provide a technique for integrating ADTs, defined in the general-purpose theorem prover Isabelle/HOL, in the domain-specific theorem prover KeY. We describe how data types, functions, and lemmas can be imported into KeY from Isabelle/HOL. Our LHB approach is not only useful for reasoning about the Java Collection Framework, but it is a general method that can also be applied to other libraries and their interfaces. We foresee that our technique can be extended to other common data types, such as trees and graphs, which provides a fruitful direction for future work.

The work using the LHB approach has opened up the possibility of defining many more functions on histories, thus furthering the ability to model complex object behavior: this we demonstrated by verifying complex and realistic client code that uses collections. The binary method takes more than 100 minutes to verify: it is hard to imagine that it can be done with the EHB approach. Moreover, we significantly simplified reasoning about the properties of user-defined functions themselves. We can fully automate verification in Isabelle/HOL with user-defined attributes modeled as a function. Further, while KeY is tailored for proving properties of concrete Java programs, Isabelle/HOL has more powerful facilities for general theorem proving. Our approach allows leveraging Isabelle/HOL to guarantee, for example, meta-properties such as the consistency of axioms about user-defined ADT functions. Using KeY alone, was problematic or even impossible.

In Chapter 7, we introduce a new history-based proof-theory that allows us to formally verify that inherited methods are correct with respect to refinements of overridden methods. Benefiting from the LHB approach, we formulate behavioral subtyping rules that can be employed to axiomatize various kinds of refinements in terms of a projection relation: from interface to interface, interface to class, and class to class. To bring these concepts to real code, we describe a simple running example that captures the key hierarchy structure and some interesting challenges in object-oriented programs, e.g. specifying interface protocols. Through this example, we demonstrate the practical applicability of our history-based refinement approach.

8.2 Future work

The research presented in this thesis achieved some interesting results and opened up several potential research directions that we leave for future work. In this section, we briefly discuss future directions related to our main topic.

In Chapter 3 and Chapter 4, we discuss the specification and verification of part of the classes and interfaces provided by the Java Collection framework. To achieve the ultimate goal of complete formal verification of Java's Collection Framework still requires a lot of effort. For example, with our novel approach, one can continue our specification and verification work on `LinkedList`, which we introduced in Chapter 3, to include methods like `retainAll` and `removeAll` that have not yet been verified. Furthermore, the verification of other classes in the Java collection framework, such as `ArryList`, remains open. While Chapter 4 focuses on the `Collection` interface, there are several other interfaces, such as, `Map`, `List`, `Set` and `ListIterator`, that warrant attention in future work.

In Chapter 6, we introduced a technique for integrating ADTs into the KeY theorem prover. We outline how data types, functions, and lemmas can be imported into domain-specific theorem prover KeY from the general-purpose theorem prove Isabelle/HOL. It is noteworthy that the translation from Isabelle/HOL to KeY is implemented manually. Our approach leverages that Isabelle/HOL guarantees the consistency of introducing user-defined ADTs and functions. We manually translate these ADTs and functions as axioms into KeY using taclet rules, and ensure that these rules can be accepted and used by KeY. This process requires the verifier to be very familiar with KeY, Isablle/HOL, JML, taclet rules, etc. From the practical perspective, an automatic tool that imports Isabelle/HOL theories into KeY based on our work could be implemented. This would further reduce manual intervention and enable full automation of the verification process.

In Chapter 7, we proposed a history-based refinement theory to verify the hierarchy structure in widely used object-oriented programs. For instance, within the Java Collection Framework, the `Collection` interface serves as a foundational component within the framework, representing a group of objects and providing a blueprint for various concrete implementations, including `List`, `Set`, and `Queue`. The more complex hierarchy structure in the `Collection` interface can be found in the class `Linkedlist` that inherits from `AbstractSequqntialList` which inherits from `AbstractList` and then inherits from `AbstractCollection` and implements the `List` interface. Benefiting from our history-based refinement theory, we can follow the hierarchy structure to systematically analyze and validate the behavioral subtyping relations between each class and interface. Besides, the refinement theory between `Iterator` and `ListIterator` is also an interesting direction, as an iterator requires a notion of ownership since its behavior depends on the history of other objects. It remains future work to apply this theory to verifying real software. Such an effort could be used to demonstrate how formal methods improve the reliability and accuracy of popular object-oriented libraries.

From a long-term perspective, it is worthwhile to consider future work related to verified code revisions and proof reuse. In Chapter 3, we discussed fixing the `LinkedList` class by explicitly bounding its maximum size to `Integer.MAX_VALUE` elements, but other solutions are possible. Rather than using integers indices for elements, one could change to an index of type `long` or `BigInteger`. Such a code revision is however incompatible with the general `Collection` and `List` interfaces (whose method signatures mandate the use of integer indices), thereby breaking all existing client code that uses `LinkedList`. Clearly, this is not an option in a widely used language like Java or any language that aims to be backward compatible. It raises the challenge: can we find code revisions that are compatible with existing interfaces and their clients? We can take this challenge even further: can we use our workflow to find such compatible code revisions, and are they also amenable to formal verification? For code reuse, many case studies in mechanic verification [1, 23, 73] indicate that the main bottleneck today is not verification, but specification. For example, the `LinkedList` case study comprised approximately 18-21 person months in total. But once the specifications were in place, after many iterations of the workflow, producing the actual final proofs took only 1-2 weeks!

Specifications typically need to be developed incrementally during the proof effort, but there is little support for such an incremental approach in KeY: minor specification changes, like adding a conjunct to a class invariant, often require to redo nearly the whole proof, causing an explosion in the amount of effort needed. This vulnerability to change arises partly from proof rules that have a very fine granularity: proof rule applications explicitly refer to the indices of the (sub) formulas the rule is applied, resulting in fragile specifications. In the current version of KeY, proofs consist of actual rule applications (rather than higher-level macro/strategy applications), and proof rule applications explicitly refer to the indices of the (sub) formulas the rule is applied to. This results in a fragile proof format, where small changes to the specifications or source code (such as code refactoring) break the proof. Moreover, the rule set used may change in different versions of KeY, limiting backward compatibility for proofs made in a different KeY version. To improve the reusability of proofs, one can develop a versioning system for proof rules that use very fine-grained proof representations. The automatic generation of high-level proof scripts by monitoring the interactions between the proof engineer and the prover is also future work. Dealing with modifications of the underlying proof system of the theorem prover while supporting resuming existing, possibly partial proofs (through the versioning system and proof gaps) is also an interesting future direction.

Bibliography

[1] de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.collection.sort() is broken: The good, the bad and the worst case. In: 27th Conference on Computer Aided Verification (CAV). Volume 9206 of LNCS., Springer (2015) 273–289

[2] de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. J. Autom. Reasoning **62** (2019) 93–126

[3] Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf. **7** (2005) 212–232

[4] Costa, D., Andrzejak, A., Seboek, J., Lo, D.: Empirical study of usage and performance of Java collections. In: 8th Conference on Performance Engineering, ACM (2017) 389–400

[5] Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M.C.J.D., de Gouw, S.: Verifying OpenJDK's LinkedList using KeY. In: 26th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 12079 of LNCS., Springer (2020) 217–234

[6] Krasner, H.: The cost of poor software quality in the US: A 2022 report. Proc. Consortium Inf. Softw. QualityTM (CISQTM) (2022)

[7] Rooney, J.J., Heuvel, L.N.V.: Root cause analysis for beginners. Quality progress **37** (2004) 45–56

[8] Mili, A., Tchier, F.: Software testing: Concepts and operations. John Wiley & Sons (2015)

[9] Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving JDK's dual pivot quicksort correct. In: 9th Conference on Verified Software, Theories, Tools, and Experiments (VSTTE). Volume 10712 of LNCS., Springer (2017) 35–48

[10] Huisman, M.: Verification of Java's AbstractCollection class: A case study. In: 6th Conference on Mathematics of Program Construction. Volume 2386 of LNCS., Springer (2002) 175–194

[11] Knüppel, A., Thüm, T., Pardylla, C., Schaefer, I.: Experience report on formally verifying parts of OpenJDK's API with KeY. In: F-IDE 2018: Formal Integrated Development Environment. Volume 284 of EPTCS., OPA (2018) 53–70

[12] de Boer, F.S., de Gouw, S., Vinju, J.J.: Prototyping a tool environment for run-time assertion checking in JML with communication histories. In: Formal Techniques for Java-Like Programs (FTfJP), ACM (2010) 6:1–6:7

[13] Kandziora, J., Huisman, M., Bockisch, C., Zaharieva-Stojanovski, M.: Run-time assertion checking of JML annotations in multithreaded applications with e-OpenJML. In: Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs. (2015) 1–6

[14] Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM (2007) 569–588

[15] Cheon, Y., Perumandla, A.: Specifying and checking method call sequences of Java programs. Software Quality Journal **15** (2007) 7–25

[16] Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time Java pro-

grams (tool paper). In: Software Engineering and Formal Methods (SEFM), IEEE Computer Society (2009) 33–37

[17] Azzopardi, S., Colombo, C., Pace, G.J.: CLARVA: Model-based residual verification of Java programs. In: Model-Driven Engineering and Software Development (MODELSWARD), SciTePress (2020) 352–359

[18] Welsch, Y., Poetzsch-Heffter, A.: A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. Science of Computer Programming **92** (2014) 129–161

[19] Visser Willem, Păsăreanu Corina S., K.S.: Test input generation with java pathfinder. In: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. (2004) 97–107

[20] Havelund K, P.T.: Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer **2** (2000) 366–381

[21] Huisman, M., Jacobs, B., van den Berg, J.: A case study in class library verification: Java's Vector class. Int. J. Softw. Tools Technol. Transf. **3** (2001) 332–352

[22] Jeffrey, A., Rathke, J.: Java Jr: Fully abstract trace semantics for a core Java language. In: Programming Languages and Systems (PLS). Volume 3444 of LNCS., Springer (2005) 423–438

[23] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M., eds.: Deductive Software Verification – The KeY Book. Volume 10001 of LNCS. Springer (2016)

[24] Huisman, M., Ahrendt, W., Grahl, D., Hentschel, M.: Formal specification with the Java Modeling Language. In: [23]. Springer (2016) 193–241

[25] Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: 17th European Symposium on Programming. Volume 4960 of LNCS., Springer (2008) 307–321

[26] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH (1995)

[27] Meyer, B.: Object-oriented software construction. Volume 2. Prentice hall Englewood Cliffs (1997)

[28] Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS) **16** (1994) 1811–1841

[29] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M., eds.: Deductive Software Verification - The KeY Book - From Theory to Practice. Volume 10001 of Lecture Notes in Computer Science. Springer (2016)

[30] Beckert, B., Giese, M., Habermalz, E., Hähnle, R., Roth, A., Schlager, S.: Taclets: A new paradigm for constructing interactive theorem provers. RACSAM **98** (2004) 17–53

[31] Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML reference manual. Unpublished manuscript, revision 2344 (2013)

[32] Paulson, L.C.: Isabelle: A generic theorem prover. Springer (1994)

[33] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic. Volume 2283 of LNCS. Springer (2002)

[34] Wenzel, M., Berghofer, S.: The Isabelle system manual (2014)

[35] Bauer, G., Nipkow, T., Oheimb, D.v., Paulson, L.C., Rasmussen, T.M., Tabacznyj, C., Wenzel, M.: The supplemental Isabelle/HOL library (2002)

[36] Biendarra, J., Blanchette, J.C., Desharnais, M., Panny, L., Popescu, A., Traytel, D.: Defining (co)datatypes and primitively (co)recursive functions in Isabelle/HOL (2016) Available at: https://isabelle.in.tum.de/doc/datatypes.pdf.

[37] Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In: 27th Symposium on Logic in Computer Science (LICS), IEEE (2012) 596–605

[38] Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M., de Gouw, S.: Verifying OpenJDK's LinkedList using KeY. In: Tools and Algorithms for the Construction and Analysis of Systems, Springer (2020) 217–234

[39] Hiep, H.A., Bian, J., de Boer, F.S., de Gouw, S.: A Tutorial on Verifying LinkedList using KeY: Proof Files (2020) Available at: https://doi.org/10.5281/zenodo.3613711.

[40] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Video Material (2020) Available at: https://doi.org/10.6084/m9.figshare.c.4826589.v2.

[41] Klint, P., Van Der Storm, T., Vinju, J.: Rascal: A domain specific language for source code analysis and manipulation. In: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE (2009) 168–177

[42] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 1 (2020) Available at: https://doi.org/10.6084/m9.figshare.11662824.

[43] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 2 (2020) Available at: https://doi.org/10.6084/m9.figshare.11673987.

[44] Bloch, J., Gafter, N.: Collection (Java Platform SE 8) (2020) Available at: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html.

[45] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3a (2020) Available at: https://doi.org/10.6084/m9.figshare.11688816.

[46] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3b (2020) Available at: https://doi.org/10.6084/m9.figshare.11688858.

[47] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3c (2020) Available at: https://doi.org/10.6084/m9.figshare.11688870.

[48] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3d (2020) Available at: https://doi.org/10.6084/m9.figshare.11688984.

[49] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3e (2020) Available at: https://doi.org/10.6084/m9.figshare.11688891.

[50] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 4a (2020) Available at: https://doi.org/10.6084/m9.figshare.11699178.

[51] Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 4b (2020) Available at: https://doi.org/10.6084/m9.figshare.11699253.

[52] Bruce, K.B., Cardelli, L., Castagna, G., Eifrig, J., Smith, S.F., Trifonov, V., Leavens, G.T., Pierce, B.C.: On binary methods. Theory Pract. Object Syst. **1** (1995) 221–242

[53] Hiep, H.A., Bian, J., de Boer, F.S., de Gouw, S.: History-based specification and verification of Java collections in KeY. In: 16th International Conference on Integrated Formal Methods, Springer (2020) 199–217

[54] Weiß, B.: Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction. PhD thesis, Karlsruhe Institute of Technology (2011)

[55] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th Symposium on Logic in Computer Science (LICS), IEEE (2002) 55–74

[56] Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: 23rd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM (2008) 213–226

[57] Banerjee, A., Naumann, D.A., Nikouei, M.: A logical analysis of framing for specifications with pure method calls. ACM Trans. Program. Lang. Syst. **40** (2018)

[58] Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Fundamental Approaches to Software Engineering (FASE). Volume 4422 of LNCS., Springer (2007) 336–351

[59] Nipkow, T.: Embedding programming languages in theorem provers. In: International Conference on Automated Deduction, Springer (1999) 398–398

[60] von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. Concurrency and Computation:

Practice and Experience **13** (2001) 1173–1214

[61] Jacobs, B., Van Den Berg, J., Huisman, M., van Berkum, M., Hensel, U., Tews, H.: Reasoning about Java classes: preliminary report. In: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. (1998) 329–340

[62] Huisman, M.: Reasoning about Java programs in higher order logic using PVS and Isabelle. PhD thesis, University of Nijmegen (2001)

[63] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In Clarke, E.M., Voronkov, A., eds.: Logic for Programming, Artificial Intelligence, and Reasoning, Berlin, Heidelberg, Springer Berlin Heidelberg (2010) 348–370

[64] Filliâtre, J.C., Paskevich, A.: Why3: where programs meet provers. In: 22nd European Symposium on Programming. Volume 7792 of LNCS., Springer (2013) 125–128

[65] Liskov, B., Zilles, S.: Programming with abstract data types. ACM SIGPLAN Notices **9** (1974) 50–59

[66] Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. Monographs in Theoretical Computer Science. Springer (2012)

[67] Leavens, G.T., Cheon, Y.: Design by contract with JML (2006) Available at: http://www.cs.utep.edu/cheon/cs3331/data/jmldbc.pdf.

[68] Darvas, A., Müller, P.: Faithful mapping of model classes to mathematical structures. In: 2007 Conference on Specification and Verification of Component-Based Systems (SAVCBS), ACM (2007) 31–38

[69] Giese, M.: Taclets and the KeY prover. Electronic Notes in Theoretical Computer Science **103** (2004) 67–79

[70] Habermalz, E.: Ein dynamisches automatisierbares interaktives Kalkül für schematische theorie spezifische Regeln. PhD thesis, University of Karlsruhe (2000)

[71] Bloch, J., Gafter, N.: Collection (Java Platform SE 7) (2010) Available at: https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html.

[72] Bian, J., Hiep, H.A., de Boer, F.S., de Gouw, S.: Integrating ADTs in KeY and their Application to History-based Reasoning about Collection: Proof Files. Zenodo (2022) Available at: https://doi.org/10.5281/zenodo.7079126.

[73] de Gouw, S., de Boer, F.S., Rot, J.: Proof pearl: The key to correct and stable sorting. J. Autom. Reason. **53** (2014) 129–139

[74] de Boer, M., de Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of jdk's identity hash map implementation. In ter Beek, M.H., Monahan, R., eds.: Integrated Formal Methods - 17th International Conference, IFM 2022, Lugano, Switzerland, June 7-10, 2022, Proceedings. Volume 13274 of Lecture Notes in Computer Science., Springer (2022) 45–62

[75] Bian, J., Hiep, H.A., de Boer, F.S., de Gouw, S.: Integrating ADTs in KeY and their application to history-based reasoning. In Huisman, M., Păsăreanu, C., Zhan, N., eds.: Formal Methods, Cham, Springer International Publishing (2021) 255–272

[76] Bian, J., Hiep, H.A.: Integrating ADTs in KeY and their Application to History-based Reasoning: Video Material. FigShare (2021) Available at: https://doi.org/10.6084/m9.figshare.c.5413263.

[77] AbdelGawad, M.A.: Why nominal-typing matters in oop. arXiv preprint arXiv:1606.03809 (2016)

[78] Leavens, G.T.: Introduction to the literature on object-oriented design, programming, and languages. ACM SIGPLAN OOPS Messenger **2** (1991) 40–53

[79] America, P.: Inheritance and subtyping in a parallel object-oriented language. In: ECOOP'87 European Conference on Object-Oriented Programming: Paris, France, June 15–17, 1987 Proceedings 1, Springer (1987) 234–242

[80] America, P.: Designing an object-oriented programming language with behavioural subtyping. In: Foundations of Object-Oriented Languages: REX School/Workshop Noordwijkerhout, The Netherlands, May 28–June 1, 1990 Proceedings, Springer (1991) 60–90

[81] Bruce, K.B., Wegner, P.: An algebraic model of subtypes in object-oriented languages (draft). ACM Sigplan Notices **21** (1986) 163–172

[82] Dhara, K.K., Leavens, G.T.: Forcing behavioral subtyping through specification inheritance. In: Proceedings of IEEE 18th International Conference on Software Engineering, IEEE (1996) 258–267

[83] Leavens, G.T.: JML's rich, inherited specifications for behavioral subtypes. In: Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006. Proceedings 8, Springer (2006) 2–34

[84] Leavens, G.T., Naumann, D.A.: Behavioral subtyping, specification inheritance, and modular reasoning. ACM Transactions on Programming Languages and Systems (TOPLAS) **37** (2015) 1–88

[85] Leavens, G.T., Weihl, W.E.: Reasoning about object-oriented programs that use subtypes. In: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications. (1990) 212–223

[86] Leavens, G.T., Weihl, W.E.: Specification and verification of object-oriented programs using supertype abstraction. Acta Informatica **32** (1995) 705–778

[87] Müller, P.: Modular specification and verification of object-oriented programs. Springer (2002)

[88] Parkinson, M.J.: Local reasoning for Java. Technical report, University of Cambridge, Computer Laboratory (2005)

[89] Pierik, C.: Validation techniques for object-oriented proof outlines. PhD thesis, Utrecht University (2006)

[90] Back, R.J., Wright, J.: Refinement calculus: a systematic introduction. Springer Science & Business Media (2012)

[91] Back, R.: On correct refinement of programs. Journal of Computer and System Sciences **23** (1981) 49–68

[92] Morgan, C.: Programming from specifications. Prentice-Hall, Inc. (1990)

[93] Goldsack, S., Kent, S.: A type-theoretic basis for an object-oriented refinement calculus. In: Formal methods and object technology, Springer (1996) 317–335

[94] Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming **62** (2006) 253–286

[95] Reus, B.: Modular semantics and logics of classes. In: CSL. Volume 3., Springer (2003) 456–469

[96] Bian, J., Hiep, H.A., de Boer, F.S., de Gouw, S.: Integrating ADTs in KeY and their application to history-based reasoning. In: Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24, Springer (2021) 255–272

[97] Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., et al.: JML reference manual (2008)

[98] Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. SIGPLAN Not. **21** (1986) 38–45

[99] Bian, J., Hiep, H.A., de Boer, F.S.: History-Based Reasoning about Behavioral Subtyping:Proof fiels. Zenodo (2024) Available at: https://doi.org/10.5281/zenodo.10998227.